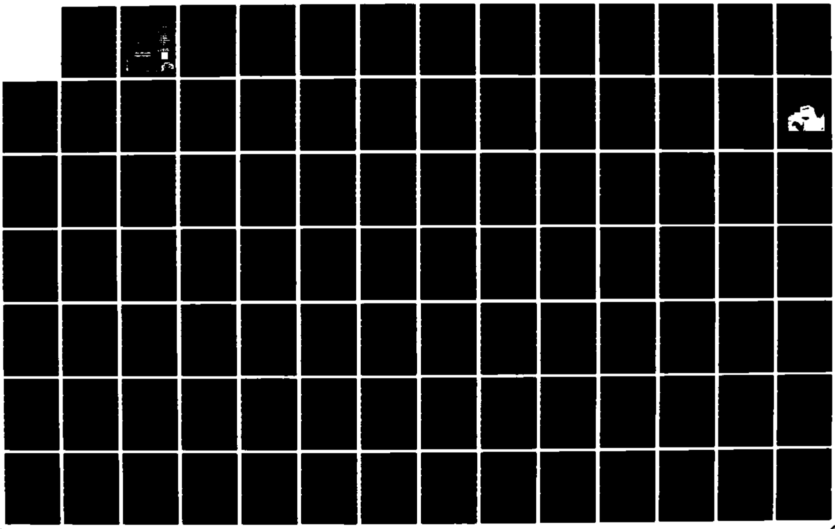
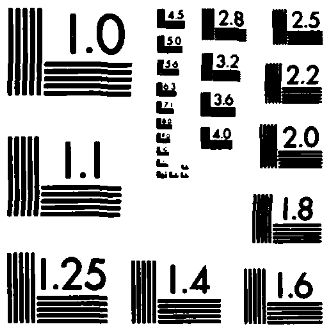
AD-A130 227 A DISTRIBUTED SIGNAL PROCESSING ARCHITECTURE(U) 1/2
MASSACHUSETTS INST OF TECH LEXINGTON LINCOLN LAB
A E FILIP ET AL. 12 MAY 83 TR-637 ESD-TR-82-178
UNCLASSIFIED F19628-80-C-0002 F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0

The work reported in this document was performed at Lincoln Laboratory, a center for research operated by Massachusetts Institute of Technology, with the support of the Department of the Air Force under Contract F19628-80-C-0002.

This technical report has been reviewed and is approved for publication.

FOR THE COMMANDER

Thomas J. Alpert

Thomas J. Alpert, Major, USAF
Chief, ESD Lincoln Laboratory Project Office

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LINCOLN LABORATORY

A DISTRIBUTED SIGNAL PROCESSING ARCHITECTURE

A.E. FILIP
Group 27

J.S. ARTHUR
Group 39

J.D. DRINAN
Group 28

A.H. HUNTOON
Group 24

D.E. KIRK
Naval Post Graduate School

J.G. VERLY
Group 27

TECHNICAL REPORT 637

12 MAY 1983

LEXINGTON MASSACHUSETTS

ABSTRACT

An architecture is described for a multi-processor implementation of real-time signal processing algorithms. A "butterfly" network is used to provide simultaneous, conflict-free interprocessor communication for multi-dimensional convolution and Fourier transformation. A hardware demonstration test-bed using four active processors was used to validate the concepts of (1) shared algorithm execution, (2) conflict-free data transfers, (3) distributed network control, (4) dynamic fault tolerance, and (5) identical software in all processors.

CONTENTS

Appendix

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 OVERVIEW

The Distributed Signal Processor (DISP) is designed to perform real–time signal processing on large data sets. Potential applications for the DISP include image processing (enhancement, restoration, segmentation, and coding), radar imaging, synthetic aperture radar, and infrared imaging (passive or active). These applications share three important characteristics. First, the data sets are large, generally multi–dimensional, with high computation rates. Second, the applications make extensive use of convolution and Fourier transformation algorithms. Finally, real–time signal processing applications generally have a regular, well defined time–line.

These characteristics are reflected in the DISP architecture. The large memory and computation requirements (often above one hundred million real operations per second) are beyond the ability of a single, programmable machine. (For comparison, the Cray–1 is rated at eighty million operations per second.) A distributed architecture, however, uses many nodal processors in parallel, which, in aggregate, can satisfy the memory and speed requirements of these applications. In addition, the distributed architecture is modular so that the number of nodal processors can be varied to suit a particular application.

Signal processing algorithms implemented on the DISP are divided into three basic steps. First, the data set is equally divided among the nodal processors. Then all nodal processors simultaneously perform the same operations on different data. Finally, all nodal processors simultaneously exchange partially–processed data through an interprocessor communication network. The second and third steps may be repeated until the task is completed.

The practical utility of a distributed approach to signal processing hinges on the selection of the interprocessor communication network. The dominance of convolution and Fourier transform algorithms implies that the interprocessor network selected for the DISP must be very efficient in handling these algorithms. The DISP uses a network which supports the data transfer requirements of these algorithms with one hundred percent efficiency, i.e., all nodal processors can simultaneously transfer the necessary data among themselves without conflict. The same network also has an efficient hardware realization with a simple distributed control strategy.

Another practical consideration for a distributed signal processing architecture is software. In the DISP, the software development task is eased because of the focus on real–time signal processing (as opposed to data processing). The highly structured time–line associated with these applications allows efficient, manual partitioning of algorithms, and results in identical software in all of the nodal processors. It also avoids the need for an operating system in each node.

There are several other advantages that are realized in a distributed signal processing architecture. The first advantage is one of economics – both initial cost and life cycle cost. The initial cost of the DISP is reduced because of its extensive use of standard microprocessor chip sets in the nodes. These components exhibit a rapidly declining cost per function because of their wide–spread use in commercial applications. The life cycle cost of a DISP–based system is reduced because its architecture uses large numbers of few board types which reduces the maintenance and logistic support costs of the processor. The reliability of the DISP is enhanced for the same reason: using large numbers of a few unique boards simplifies the introduction of fault tolerance into the design.

A third advantage of the DISP approach is that of flexibility. While dedicated, hardwired signal processors may provide a comparable computational capability, they are unable to adapt to a changing mix of algorithms that frequently occurs as a system moves from its inception to its use in the field. The DISP is capable of being reprogrammed to satisfy the evolutionary needs of a system. This also extends the useful life of the signal processor and contributes to a lower life cycle cost.

Finally, the microprocessors and their peripherals used in the DISP architecture will rapidly benefit from technology transfer from commercial developments in Very–Large–Scale–Integration (VLSI) and from the Very–High–Speed–Integrated–Circuit (VHSIC) program sponsored by the Department of Defense. Commercial VLSI products will be directed toward expanding or supporting existing microprocessor product lines. The large commercial market for these products insures a long–term commitment to products which should be of direct benefit to the DISP. Similarly, components now being developed for the VHSIC program are either improved microprocessors or components which can be integrated into a microprocessor–based design.

1.2 ORGANIZATION

Chapter 2 provides an overview of the main elements of the DISP architecture including the nodal processors, the timing and fault manager, and the butterfly and sparing networks. The properties of the butterfly network are examined in detail in Chapter 3. Chapter 4 describes the DISP test–bed which was constructed to verify the basic distributed signal processing concepts.

The appendices contain a variety of supporting material. Appendix A contains a derivation of the multi–processor FFT coefficients. Appendix B contains the logical equations describing the operation of the butterfly switch. Appendix C provides a detailed description of the time–line when executing a one–dimensional, multi–processor FFT on the test–bed. The transfer tables used to control data transfers among processors are described in Appendix D, while the mailbox concept used to pass control and diagnostic information is described in Appendix E. Appendix F describes the reconfiguration process in response to nodal processor faults. Finally, Appendix G describes some of the utility software that was developed to support the operation of the test–bed.

Chapter 2

ARCHITECTURE

2.1 INTRODUCTION

In the taxonomy of distributed processing systems, the DISP would be classified as a single–instruction–multiple–data (SIMD) type machine. This is consistent with the basic operating mode of the DISP wherein each nodal processor performs the same operation on different data segments. However, the DISP is not limited to operate as a SIMD machine. The butterfly network will also support "multi–SIMD" operation in which groups of processors operate as individual SIMD machines. Further, if "group" is defined as a single nodal processor, then the DISP becomes a multiple–instruction–multiple–data (MIMD) machine.

As shown in Figure 1, the major architectural features of the DISP are: (1) a set of nodal processors, (2) a "butterfly" interconnection network, (3) a timing and fault manager, and (4) a sparing switch network. These subsystems are described in the following sections.

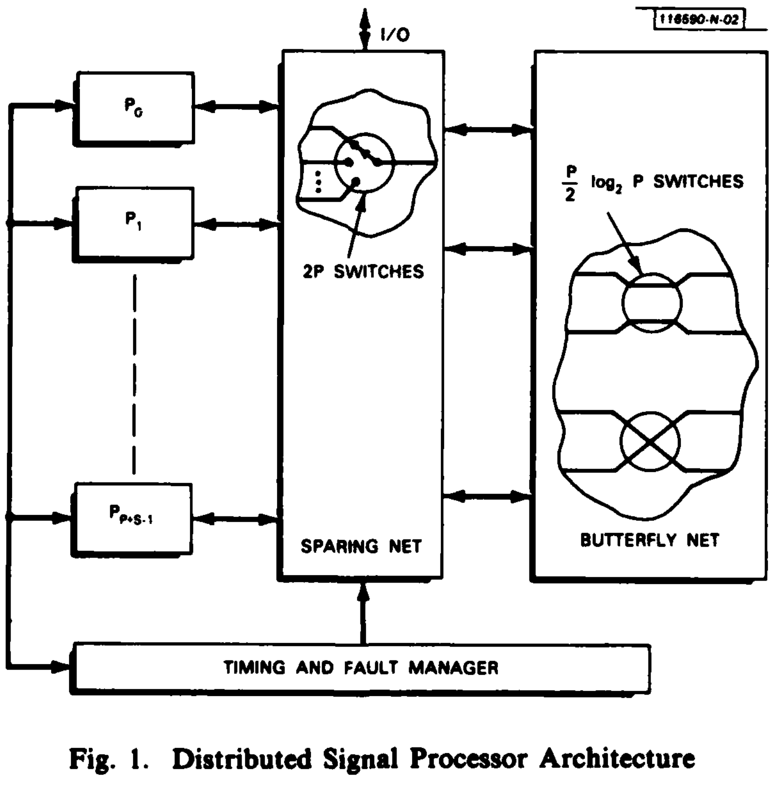


Fig. 1. Distributed Signal Processor Architecture

2.2 NODAL PROCESSORS

Figure 1 shows a total of P+S programmable nodal processors (NP) connected to the sparing switch network. The processors are completely independent of one another. There is no shared memory or common clock in the DISP which would constitute a reliability choke point. Only P of the processors are active at one time, with S processors available as warm spares. This arrangement yields substantial improvement in the mean time between failure (MTBF) for a very modest increase in total hardware. (Section 2.5)

Each nodal processor contains a triad of processors plus local memory as shown in Figure 2. The elements of the triad are: (1) a nodal manager (NM) which oversees the operation of the node; (2) a signal processor (SP) which provides the high-speed arithmetic capability in the node; (3) the input-output processor (IOP) which manages the I/O traffic between the node and the butterfly network and/or the outside world.

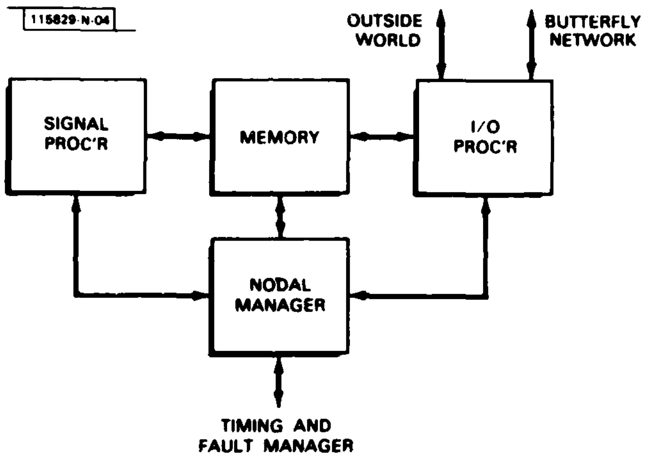


Fig. 2. Nodal Processor Block Diagram

While the modest computational load on the NM can be satisfied with a standard microprocessor (e.g., MC68000) plus support hardware, the SP and IOP require more sophistication. The SP is currently envisioned as a bit-slice microcomputer (e.g., AMD 29500 family) which is capable of performing a real operation (multiply or add) in 80-100 nsec. The IOP is currently viewed as a collection of DMA (Direct Memory Access) controllers, perhaps augmented with AMD's 29116 controller for address calculation.

In addition to the processor triad, the local memory will be organized as a multi-segmented memory containing a total of 128 Kbytes. The total number of integrated circuits (ICs) required for the nodal processor is predicted to be approximately 400.

2.3 BUTTERFLY NETWORK

The butterfly network (BFN) occupies the right-most block of Figure 1, and provides the means for interprocessor communication in the DISP. It consists of a set of two-input-two-output switches that can assume *either* of two states (crossed or straight) as shown in Figure 1. The network contains a total of P/2 logP such switches.[1] The interconnection of the switches for eight active processors is shown in Figure 3. Note the strong resemblance of the interconnection diagram to the flow diagram of the standard, radix-2, fast Fourier transform algorithm. This similarity not only prompted the name "butterfly" for the network, but it also provided the intuitive basis for the (correct) assertion that the network can support distributed FFT calculations without conflict.

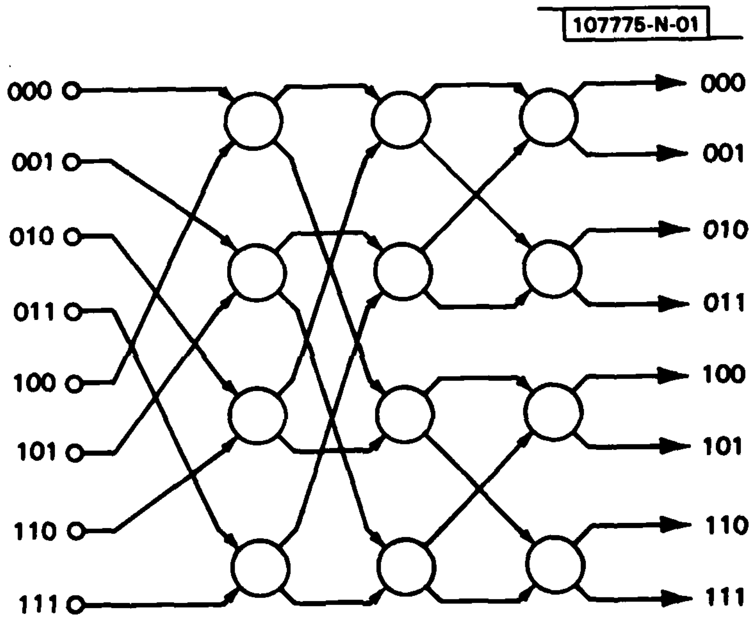


Fig. 3. Eight-Port Butterfly Network

The butterfly network was selected for use in the DISP for three reasons. First, the BFN can provide simultaneous, conflict-free communication among all nodal processors for the signal processing algorithms of interest. Second, the BFN can be controlled through a simple, distributed control algorithm. Finally, the BFN has an efficient hardware implementation.

The properties of the BFN will be described in detail in Chapter 3.

---

[1] Unless otherwise stated, all logarithms are base two (2).

2.4 TIMING AND FAULT MANAGER

The timing and fault manager (TFM) shown at the bottom of Figure 1 is responsible for: (1) interpreting fault detection reports and reconfiguring the sparing switches accordingly, and (2) task synchronization of the nodal processors. It will be implemented by using the nodal manager hardware portion of the nodal processor design.

The fault detection and interpretation function requires low-rate, bi-directional communication with the nodal processors which is provided by a serial link. The nodal processors use the serial link to inform the TFM of diagnostic results, while the TFM uses the link to assign a logical identity to each processor. The mapping between logical and physical identity will change dynamically during system operation based upon diagnostic results. The TFM must also have a control input to the sparing switch network to determine which of the P+S nodal processors actually have access to the butterfly network.

The second function of task synchronization is also achieved by means of the serial link described above. Task synchronization is occasionally required at system start-up, or after system reconfiguration. Note, however, that because of the self synchronizing capability of the butterfly network, the DISP would continue to operate in the event of a TFM failure until the system encountered an additional fault in one of the nodal processors.

2.5 SPARING SWITCH NETWORK

The sparing switch network shown in the middle of Figure 1 implements two functions in the DISP. It controls which of the P+S nodal processors have access to the butterfly network. It also houses the hardware for routing data between the P active nodal processors and the outside world. There are 2P sparing switches. On the input side of the BFN, the switches provide a (S+1):1 multiplexer function , while on the output side they provide a 1:(S+1) demultiplexer function.

The sparing switch provides the basic fault tolerance capability for the DISP. By adding a small number (S) of spare processors to the system, significant improvements in the system MTBF can be obtained. Figure 4 illustrates the predicted MTBF versus the number of spare nodal processors for systems with 4, 32, and 256 active processors. The results are based upon the byte-wide data path used in the DISP test-bed, with an assumed IC failure rate of once per ten million hours. The curves take into account the varying complexity of the sparing switch and butterfly net as the number of active and spare processors changes. Each nodal processor is assumed to contain 400 ICs, while each byte-wide butterfly switch contains 8 ICs based on the test-bed design described below. The sparing switches each contain 0, 6, 12, or 18 ICs depending on the number of spare processors. Note that adding more than two spares has only marginal benefit.

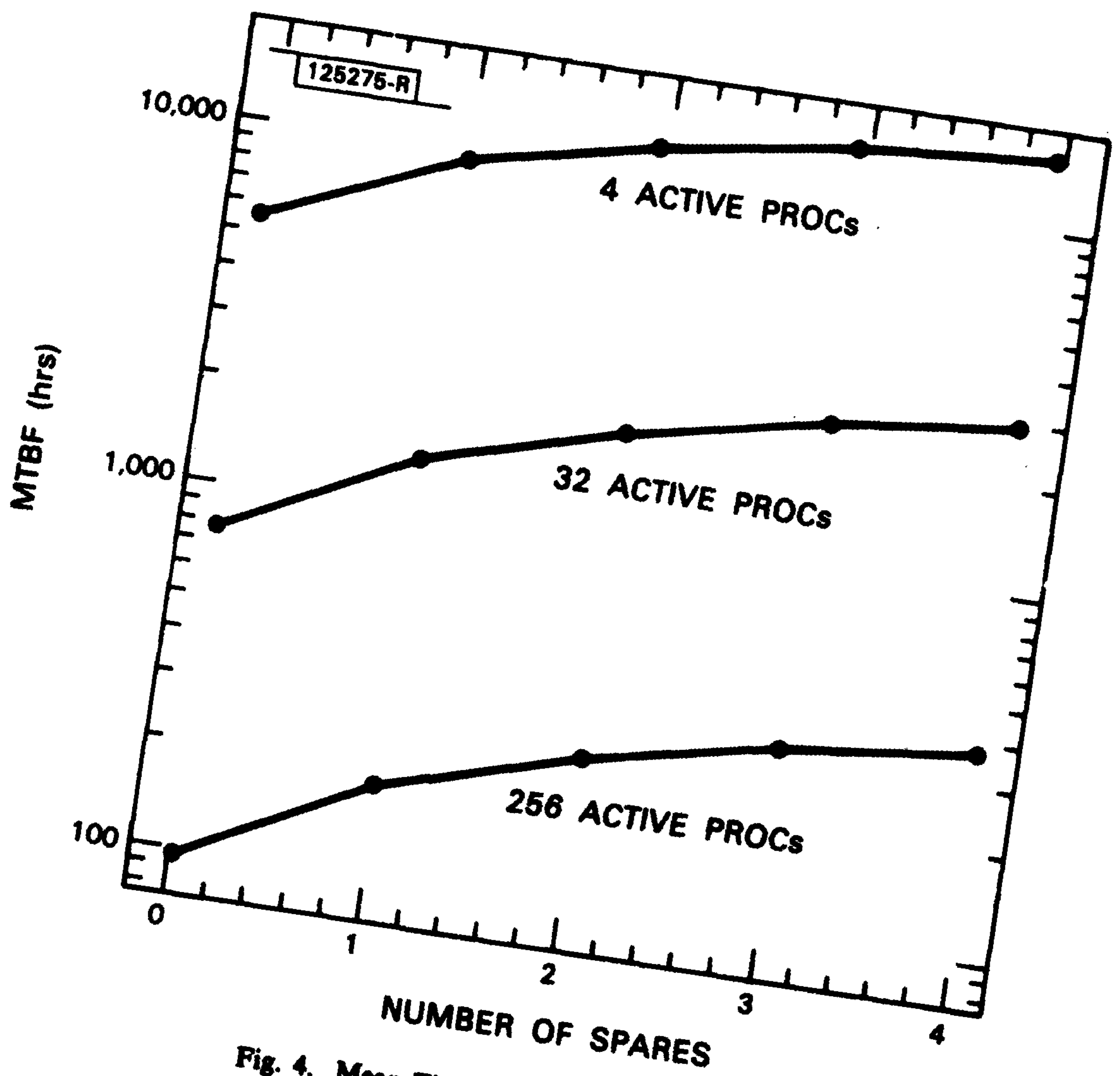


Fig. 4. Mean Time Between Failure

Chapter 3

BUTTERFLY NETWORK PROPERTIES

3.1 INTRODUCTION

The butterfly network used in the DISP owes its name to the pictorial and topological similarity it shares with the radix-2 butterfly used in the fast Fourier transform (FFT) algorithm. The correspondence between the butterfly network and the FFT flow diagram extends beyond their graphic similarity. As we shall see in Section 3.3, the butterfly network can support simultaneous, conflict-free data transfers among all nodal processors during the execution of the FFT. However, before describing the properties of this network, we shall review its genealogy and topological equivalents.

The origin of the butterfly network can be traced back almost thirty years to Clos's pioneering work in multi-stage switching networks for use in telephone systems [1]. A diagram of such a network is very similar to the flow diagram of a mixed-radix FFT algorithm [2]. They are more general than a butterfly net because they will support arbitrary permutations among the nodal processors. In 1964, Benes [3] described a highly regular network based on the two-state butterfly switch that is obtained by reflecting the BFN about its last stage. The Benes net contains a total of P(log(P)-0.5) butterfly switches, and, like Clos's network, it supports arbitrary permutations. These networks never achieved widespread use because of the difficulty in devising efficient control algorithms. For example, Opferman and Wu [4] described a control algorithm for the Benes net which requires on the order of PlogP operations to set the switches to achieve a desired permutation.

During the mid-seventies, a number of papers were published describing networks that are topologically equivalent to the BFN, such as the omega net [5], the flip net [6], the indirect binary n-cube [7], and others [8-12]. Rettberg, et al [13], were apparently the first to use the term "butterfly network". Subsequently, many authors [14-18] noted the topological equivalence of these switching networks. Thus, our use of the butterfly network in computer communications is not unique. What is unique is its application to real-time signal processing.

A comparison of the number of distinct connections among P processors (P!) with the number of states attainable in the BFN ($P^{P/2}$) reveals that the BFN cannot implement a large fraction of the possible permutations. The possible conflicts that occur throughout the network result in an average normalized bandwidth or efficiency that is substantially less than unity [19,20]. However, this limitation exists only for those applications where communication among processors occurs randomly as in a data processing or data communication context. Designers have tried to mitigate this problem by adding a store-and-forward capability into the network switches. However, signal processing algorithms, such as multi-dimensional convolution and Fourier transformation, tend to have a very simple and regular structure which requires many fewer connections. The result [7,15] is that the BFN has an efficiency of unity for the signal processing algorithms of interest, and we can thus use a very simple, memoryless, circuit-switched network structure.

Potential DISP applications are dominated by multi-dimensional convolution (correlation) and Fourier transformation. For that reason, the following two sections describe how these algorithms would be implemented on the DISP with the butterfly interconnection network. Finally, Section 3.4 describes the distributed control feature of the BFN. A detailed description of the hardware implementation of the BFN will be deferred to Chapter 4.

3.2 CONVOLUTION

Consider one-dimensional convolution (or correlation). Assume that an N-point sequence is to be convolved with a Q-sample impulse response, where $Q << N$. (If the last condition isn't satisfied, then fast convolution using FFTs is more efficient.) If the N samples are equally distributed among the P active processors, the convolution calculation can proceed in parallel in all P processors until the Q-point kernel has been shifted to the end of the data resident in each processor. At that time, in order to complete the calculation, Q-1 points must be transferred from processor i to processor [i+1] where the brackets indicate modulo-P and i ranges from 0 to P-1. This is simply a simultaneous, cyclic shift of the data by one processor as shown in Figure 5. After the data transfer, the convolution on the remaining Q points in each processor is completed. Note that if a linear (as opposed to circular) convolution is desired, then the "end" processor, P_0 does not continue its calculation after the data transfer.

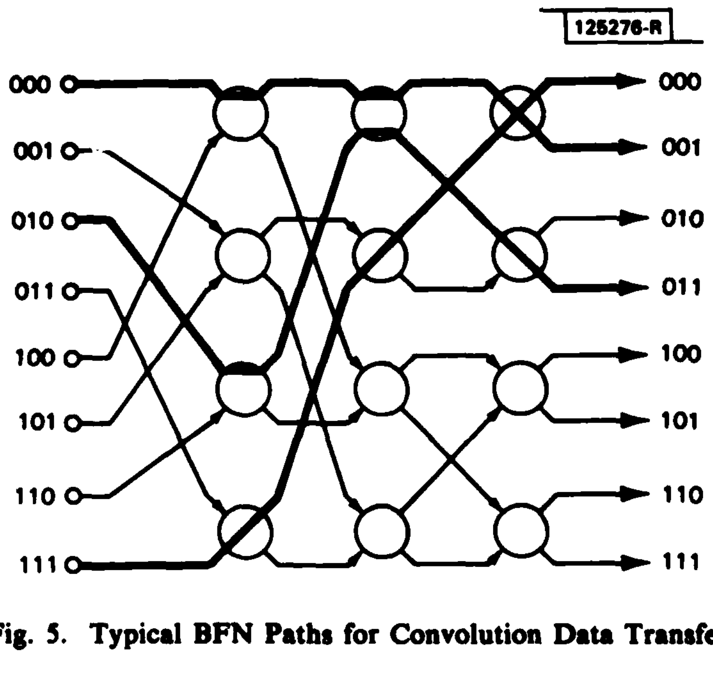


Fig. 5. Typical BFN Paths for Convolution Data Transfer

The same data transfer strategy extends directly to multi–dimensional convolution (or correlation). Once again, a cyclic shift of the data by one processor is required, but now Q–1 rows (two dimensions) or planes (three dimensions) are transferred instead of Q–1 points.

3.3 FAST FOURIER TRANSFORM (FFT)

3.3.1 One–dimensional FFT

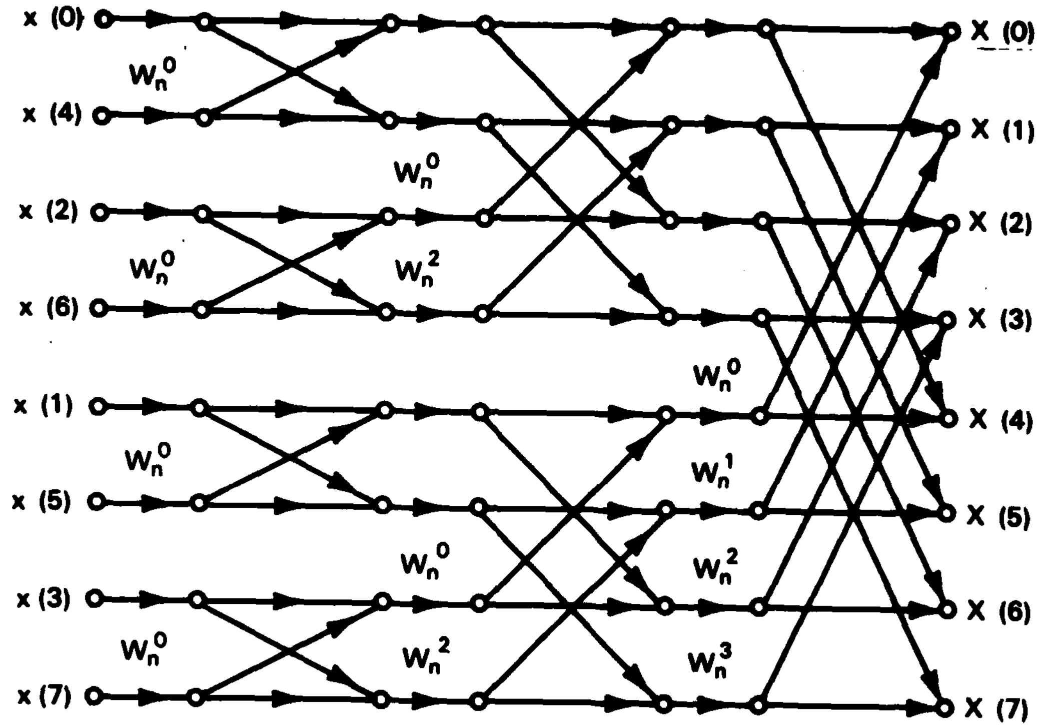
The data transfers through the BFN needed to support FFTs are slightly more involved than for convolution. Again assume that an N–sample sequence has been equally divided (after bit–reversal) among the P processors. As shown in Figure 6 for an 8–point FFT, a standard radix–2, decimation–in–time, in–place FFT algorithm combines data samples that are separated by 1, 2, 4, 8, etc. address locations as one moves through the stages of the algorithm. Eventually, the required address separation will exceed the size of the data block held by a single processor, and a data exchange must be executed. The total number of transfers needed is independent of N and is equal to logP.

Figure 7 illustrates how four processors (P_0 – P_3) would execute a 64–point, one–dimensional FFT. After dividing the data equally among the four nodal processors, the first four stages of the six–stage FFT can be computed locally. At this point, the required data separation (16 samples) exceeds the data contained within a single processor and a transfer will have to occur.

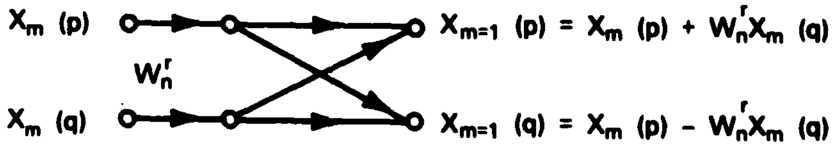
The data transfer has a very regular structure. As shown in Figure 7, the last half of the 16 samples in processor 0 are exchanged with the first half of the data in processor 1. Simultaneously, processors 2 and 3 exchange half of their data. After the data transfers are completed, the fifth stage of the FFT can be computed. Once again, a data transfer is needed before proceeding. The second transfer is similar to the first in that processors 0 and 2 exchange half of their data while processors 1 and 3 simultaneously exchange half of theirs. After the transfers, the sixth and final stage of the 64–point FFT is completed.

Note that upon completion of the calculation, the data is not normally ordered as one would expect with a single processor FFT algorithm. Instead, the data is in "pseudo–normal" order. For most applications, this is of no consequence since the data can be read in sequence from P_0...P_3, P_0...P_3. If necessary, the data can be restored to normal order with additional passes through the BFN.

The general characteristics of the FFT data transfers are: (1) logP transfers are needed, and (2) each transfer involves half of the data samples in a pairwise exchange that can be described in terms of the "cube" function defined by Siegel [21]. Figure 8 illustrates the BFN configuration to support pairwise exchange between processors 0 and 2 in an eight–processor network.



A BUTTERFLY IS


X_m (p)
X_m (q)
W_n^r
$X_{m=1}$ (p) = X_m (p) + $W_n^r X_m$ (q)
$X_{m=1}$ (q) = X_m (p) − $W_n^r X_m$ (q)

WHERE $W_n^r = \epsilon^{-j\,(2\pi/n)\,\bullet\,r}$

115830-N-01

Fig. 6. Eight–Point FFT

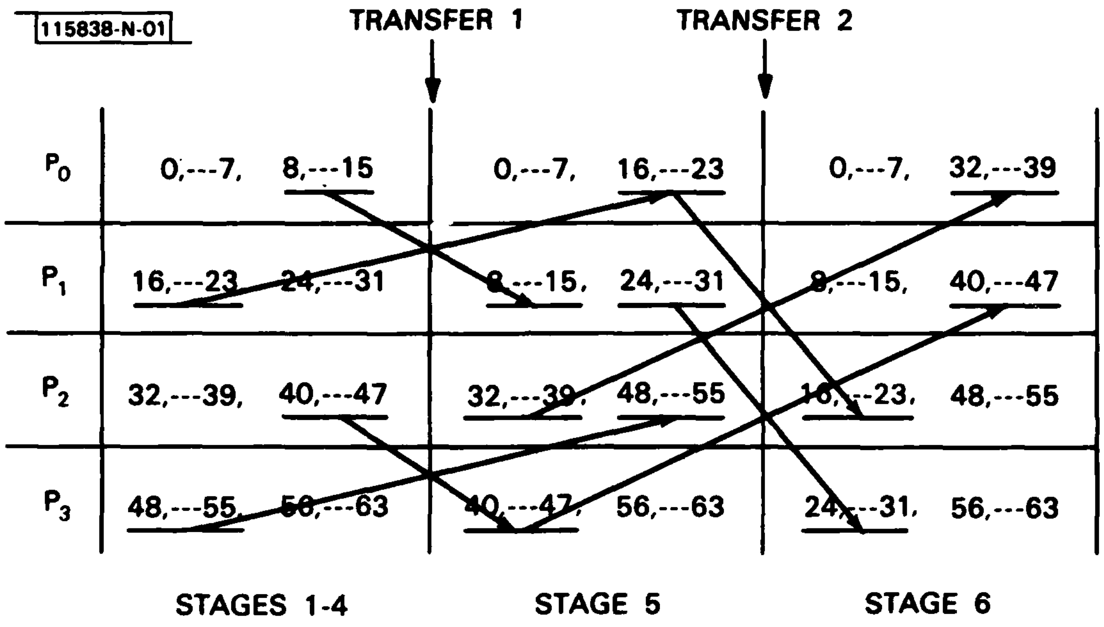


Fig. 7. 64–Point FFT on the DISP

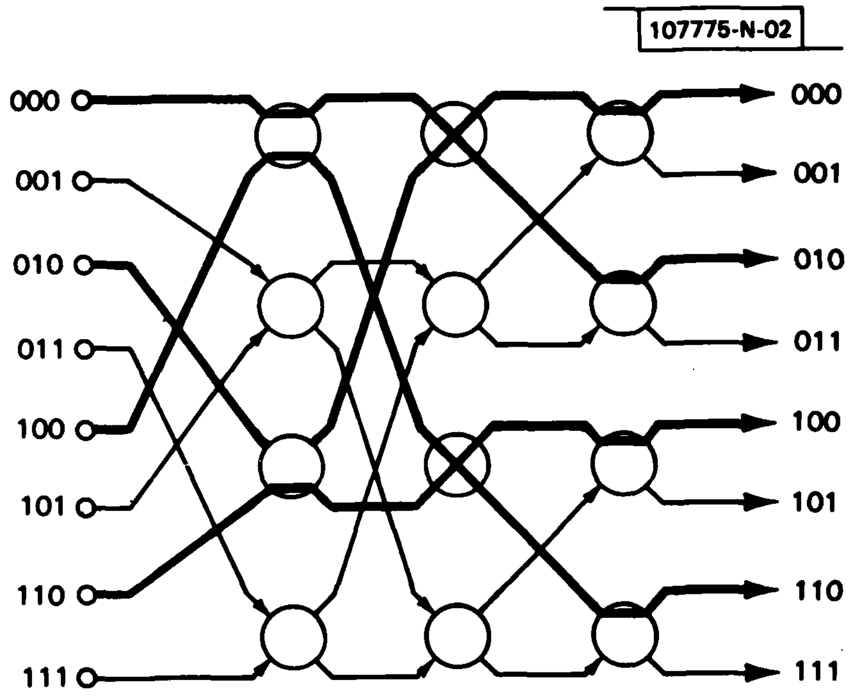


Fig. 8. Typical BFN Paths for FFT Data Transfers

3.3.2 Multi–dimensional FFT

When considering multi–dimensional problems, the FFT is richer than convolution in that new transfer strategies become available. Table 1 compares four different algorithms for computing a two–dimensional FFT on the DISP [22]. The simplest approach, called the sequential row–column (SRC), is to divide the NxN data field equally by row among the P processors so that each processor contains N x N/P samples. The FFT of the individual rows can be computed internally in each processor. The column transforms, however, require the same data transfers as described in the one–dimensional case. Once again, there are log(P) transfers, and at each transfer half of the data field is moved through the BFN.

An algorithm called the interleaved row–column (IRC) [23] orders the calculation somewhat differently. In its simplest form, one computes the first stage of all the rows, then the first stage of all the columns, then the second stage of all the rows, and so on. A more useful form of the algorithm first computes the internal stages of all the columns. A transfer of several rows is initiated while the entire transform of the stationary rows is computed. The result is that data transfers can be spread over a much longer interval than for the SRC with a consequent reduction in data rate through the BFN.

A third algorithm, called the vector–radix (VR) [24] offers another alternative that not only reduces the data transfer rate, but also the computation rate.

Finally, a fourth algorithm for the two-dimensional FFT is based upon doing "corner-turning" (CT) with the network. As with the SRC, the individual nodal processors first perform the row transforms. Then, P-1 passes are made through the network to effect the "corner turn" or matrix transpose operation. Each transfer involves $(N/P)^2$ points per processor. After the transfers are complete, the nodal processors compute the column transforms.

TABLE 1

COMPARISON OF FOUR, 2-D FFT ALGORITHMS

	SRC	IRC	VR	CT
No. Transfer Epochs[1]	q	q	q	P-1
No. Words/Epoch/ Proc'r	$N^2/2P$	$N^2/2P$	$N^2/2P$	$(N/P)^2$
Normalized Time[2] for Net I/O	.5q/r	.5(1+q/r)	q/r	.5
Normalized Net[3] Bandwidth	1	q/(r+q)	2/3	2(P-1)/rP

[1] q=log(P); r=log(N)

[2] One time unit equals the time to compute all butterflies for an NxN field divided by the number of processors.

[3] Normalization constant equals radix-2 butterfly computation time.

3.4 DISTRIBUTED NETWORK CONTROL

As mentioned in Section 3.1, the precursors to the butterfly network were seldom used because of the difficulty in determining the setting of the individual switches. Indeed, the Benes net uses a separate minicomputer solely for this purpose. Such an approach for the DISP is unacceptable from a reliability standpoint because a single-point failure in the control minicomputer would shut down the entire network.

Fortunately, as several authors have noted, [7,10,14,17] the butterfly network lends itself to a simple, distributed control algorithm. The algorithm is based upon the fact that there are as many columns or stages in the butterfly network as there are bits in the binary representation of the nodal processor address. Thus, as shown in Figure 3 for an eight-node system, three bits are needed to describe a nodal address, which is equal to the number of stages in the network.

One implementation, termed destination addressing, is shown in Figure 9. In this case, each bit in the binary representation of the destination address is examined by the corresponding stage of the BFN. If the bit is "0", then the upper switch output is selected. If the bit is "1", the lower output is selected.

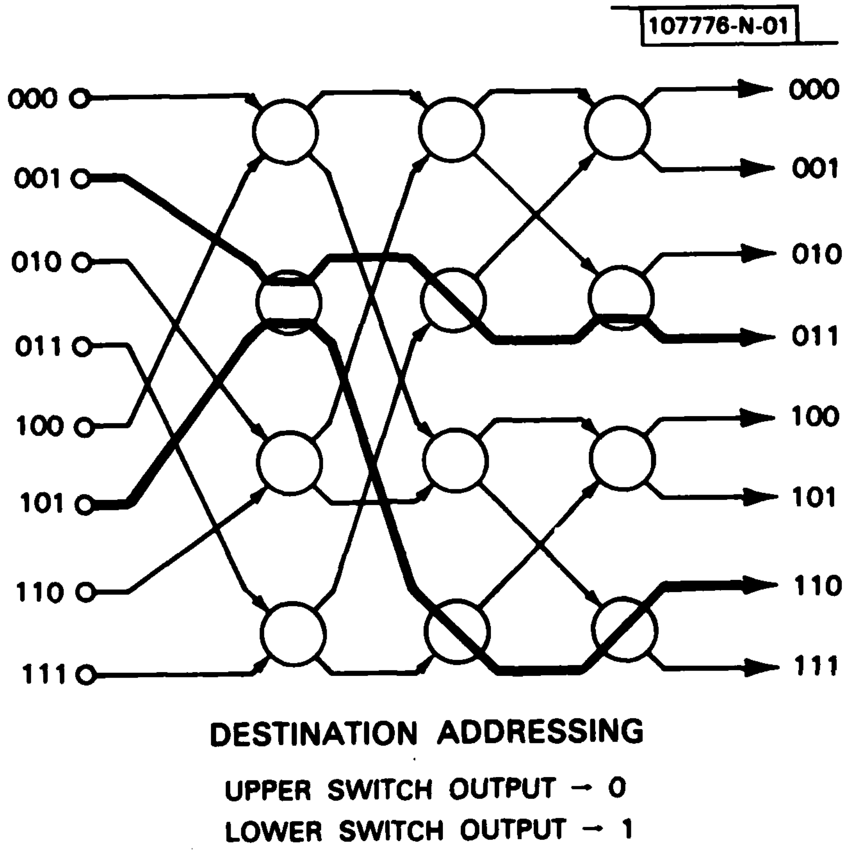


Fig. 9. Distributed Control with Destination Addressing

With a slight modification, this algorithm can be described more naturally in terms of the "crossed" and "straight" states of the butterfly switch element. First, a modified address is generated by computing the Exclusive-OR of the destination address and source address. Once again, each bit of the *modified* address is examined by the corresponding stage of the BFN. If the bit is "1", then the switch assumes the crossed state. If the bit is "0", then the "straight" state is selected.

The modified address form of the distributed control algorithm has an additional advantage in that the destination processor is able to recover the source address by computing the Exclusive-OR of the received address with its own address. Thus the

destination processor can verify that the appropriate source processor is trying to establish a link through the BFN, which is useful diagnostic information.

Chapter 4

DISP TEST-BED

4.1 INTRODUCTION

The value of any new technology, like DISP, can best be measured if a full system is built and operated in a real-time environment. However, there are fundamental concepts which can be verified with a mini-DISP system and which can be extrapolated to a full-size system. For this reason, a DISP test-bed was implemented at the Laboratory (Figure 10).

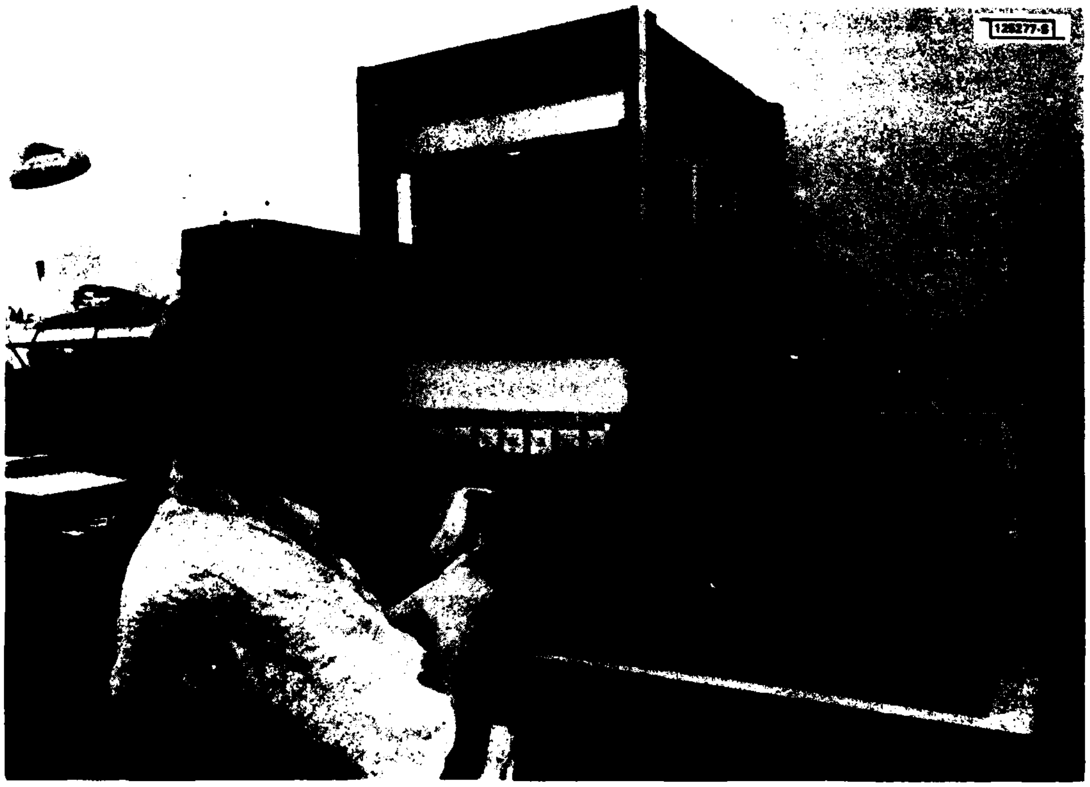


Fig. 10. DISP Test-Bed

The test-bed provides a means for validating five DISP concepts: (1) simultaneous, parallel algorithm execution, (2) simultaneous, conflict-free data transfer, (3) distributed network control, (4) dynamic fault detection and reconfiguration, and (5) identical software in all nodes. It does not currently include *real-time* computation or I/O capability, although these are planned for the future.

Most of the hardware and all of the support software were purchased from Motorola Inc. Notable hardware exceptions are the butterfly network and the sparing-switch networks which were designed at the Laboratory using commercial integrated circuits. Figure 11 shows the block diagram of the test-bed. The block diagram is equivalent to Figure 1 with P=4, and S=1. That is, there are four active nodal processors and one spare processor. Consequently, the BFN has four butterfly switches arranged in two columns or stages. All nodal processors and the TFM are commercial, single-board computers (Motorola VM01A Versamodules). The VM01A assumes the role of nodal manager, I/O processor, and signal processor, albeit at greatly reduced speed. Parallel ports on the VM01A computers connect all nodes to the BFN via the sparing switch network. Control for the sparing switch network comes from another VM01A acting as timing-and-fault manager (TFM).

The Laboratory's Amdahl 470 computer was used to download data sets and programs into the VM01As and to upload computed results for display. No separate I/O hardware was built for handling data sets to and from the outside world in real-time.

Before a distributed FFT example could be demonstrated, a number of tests had to be developed for the VM01As. For example, tests were developed to validate the performance of each port, bus, timer and control on the Motorola boards. Similarly, tests were developed to make proper use of Motorola's on-board Monitor program (VERSABUG). Finally, a series of maintenance tests and techniques were developed, first to permit step-by-step subsystem integration from 1 to 5 nodal processors with the network and TFM, and second to debug hardware and software flaws in the system. There is a new dimension involved with debugging systems of multiple general-purpose computers that does not exist with a single computer, and testing techniques which are developed on the test-bed are probably as valuable as the other DISP concepts verified in the test-bed.

The following sections describe the DISP test-bed in detail. Sections 4.2–4.5 describe the hardware, including the nodal processors, butterfly switch, sparing switches, and timing-and-fault manager. Sections 4.6–4.8 describe the software developed for the test-bed, including the one-dimensional, multi-processor FFT, the TFM and NP programs, and the software development tools. Finally, Section 4.9 contains examples of one-dimensional, multi-processor FFT calculations performed on the test-bed, and also presents some performance measures for the test-bed.

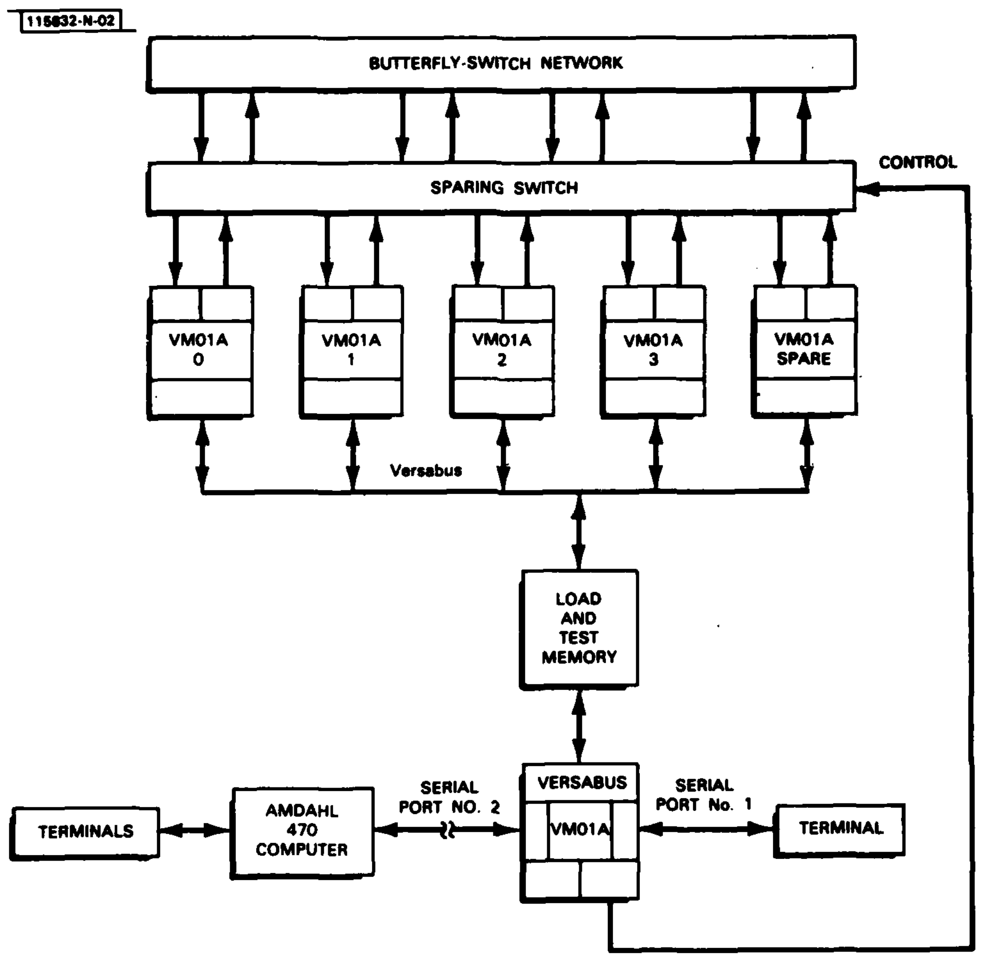


Fig. 11. Block Diagram of DISP Test–Bed

4.2 NODAL PROCESSORS

A commercial, single–board computer was used as the NP and TFM. The Motorola VM01A mic"ocomputer was selected because it contains a reasonable combination of processor (MC68000), on–board memory, and I/O capabilities. In addition, other groups in the Laboratory were already using the board, and this proved to be a valuable source of experience and software. The main features of the VM01A are listed in Table 2, while the features of the MC68000 are listed in Table 3.

The VM01A nodal processors currently perform the signal processing function (non real–time) in addition to nodal I/O and nodal management duties. The next–

generation NP implementation will include enhanced network and "outside world" I/O capability, allowing the VM01A board within each multi-board NP to function more nearly as a true nodal manager. The addition of real-time nodal signal processing capability is planned as a future enhancement.

TABLE 2

FEATURES OF VM01A MICROCOMPUTER

MC68000 MPU (8 MHz)
Versabus interface
32 Kbyte dynamic RAM with byte parity
ROM/EPROM capacity to 64 Kbytes
Two serial I/O ports (RS-232C, RS-422)
Four parallel byte-wide I/O ports, plus handshake lines
One triple 16-bit programmable timer
EPROM-resident monitor/debugger (Versabug)

TABLE 3

FEATURES OF MC68000 MICROPROCESSOR

16/32-bit (external/internal) general purpose architecture
16 Mbyte direct addressing range
17 multi-function 32-bit registers
56 instruction types
Operations on 5 data types
Memory-mapped I/O
14 addressing modes

4.3 BUTTERFLY SWITCHES

Each butterfly switch includes a controller and the requisite crosspoint steering for both data and control signals in accordance with its straight/crossed convention. Figure 12 shows the two defined states of the butterfly switch along with the corresponding logical model. The steering convention is STATE=1 for crossed connections and STATE=0 for straight connections. This butterfly switch definition does not include broadcast capability.

While data passes through the network uni-directionally, the network control signals traverse the network in both directions. These controls include separate request and acknowledge signals for network configuration and for data exchange, all of which are subject to butterfly steering. Therefore, a byte-wide butterfly switch would require at least 48 input-output pins, as indicated in Figure 13.

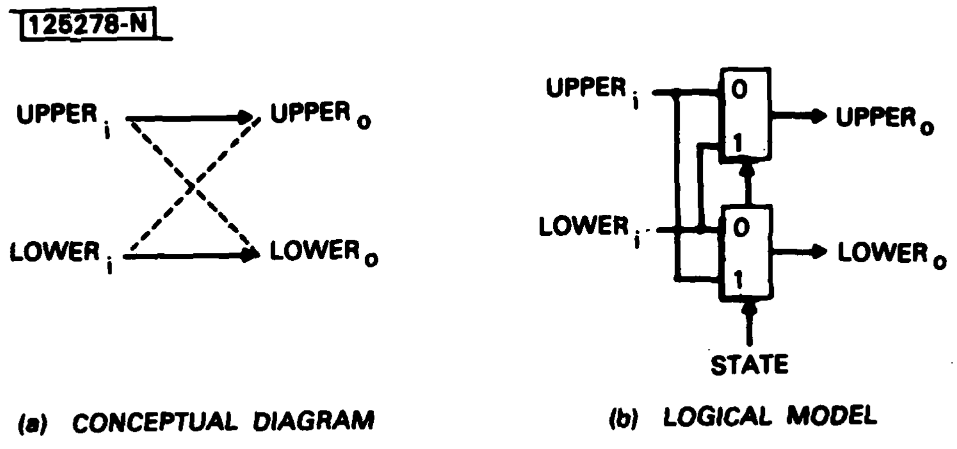


Fig. 12. Butterfly Switch

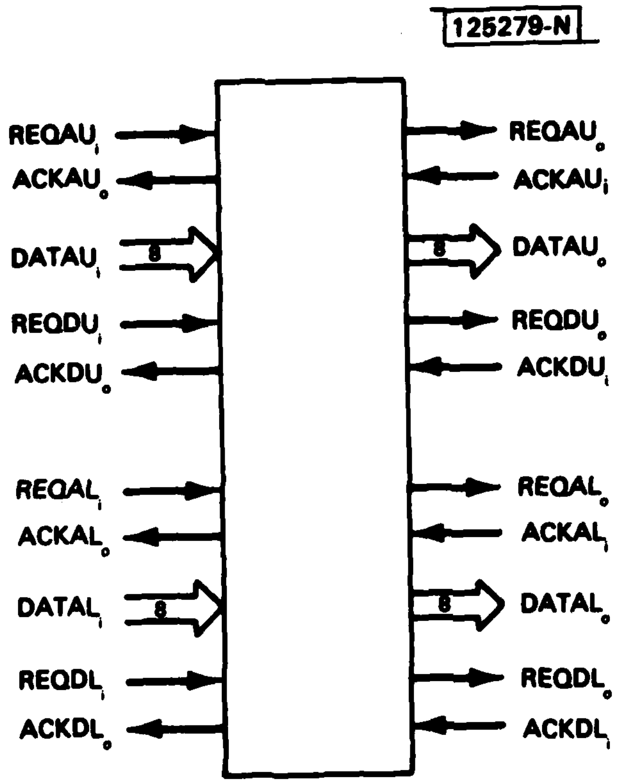


Fig. 13. Butterfly Switch Signals

The signal naming conventions of Figure 13 delineate request, acknowledge, address and data lines with the appropriate suffix. In general, any signal is denoted as wwwxyz, where www=REQ or ACK for request or acknowledge, x=A or D for address or data, y=U or L for upper or lower, and z=i or o for input or output. For example, $REQAU_i$ is an upper channel network address input request, whereas $ACKDL_o$ is a lower channel data transfer acknowledge output.

The four-node butterfly network in the test-bed contains four butterfly switches which operate asynchronously with respect to each other and also with respect to the attached NPs. Each source processor (supplying data to the network) secures its path through the network using the destination addressing technique described in Section 3.4. After the network configuration process is complete, all processor network connections are maintained by not changing the address handshake lines until another network setting is needed.

4.3.1 Butterfly Switch Control

The network control mechanism, while described in terms of a simple four-node network, can be extended to distributed systems having large numbers of NPs. Separate addressing and data transfer time intervals are assumed, and it is expected that the time invested in addressing the network will usually be amortized over extended periods of data transfer activity. When the network is being addressed, the configuration of butterfly switches occurs level-by-level across the network in a ripple fashion. The control logic within each butterfly switch stores the STATE of the switch (straight vs crossed). With the aid of some wait loops within the intelligent I/O ports of attached processors, it is also possible to distinguish between a network address request which has had inadequate time for completion, and a "drop-out" fault, wherein some butterfly switch has inadvertently changed state after having been configured.

The network addressing process depends on hardware which responds to specific REQA and ACKA edge transitions and a round-trip "ripple-through" of control signals, where request signals propagate left-to-right from the processor output ports, followed by right-to-left propagation of acknowledge signals initiated at the processor input ports. Because a network of virtually any size may be constructed (subject to delay constraints) using identical butterfly switches as building blocks, a description of the operation of a single butterfly switch and its control logic is sufficient.

The four control modes for the butterfly switch are determined by the address request and acknowledge signals as shown in Table 4. In IDLE mode, all processor output ports place a destination address on the data bus and then raise their request lines. Level by level across the network enroute to the processor input ports, each butterfly switch extracts an address bit per channel from the data bus. From the perspective of an individual butterfly switch, an upper or lower channel address bit (AU or AL) in the presence of an address request ($REQAU_i$ or $REQAL_i$) is interpreted as a "straight request" if $A_{U,L}=0$ and a "cross request" if $A_{U,L}=1$. If the butterfly switch is in IDLE mode prior to the address request, its internal STATE register will be updated, allowing requests to propagate further into the network.[2]

---

[2] Some additional restrictions on the updating of butterfly switches will be discussed later in conjunction with the controller design.

As noted in Table 4, a butterfly switch channel is in ADDRESS mode when a request has caused a STATE update, but no corresponding acknowledge has arrived. As individual source processor requests cross the network and arrive at the destination ports of recipient processors, address acknowledge signals (ACKA) are issued which traverse the just-established paths through the network (from right to left) back to the NP source ports. Now, from the perspective of the source ports, ACKA=REQA=1, and the network is in DATA mode and available. Note that the data and control lines are steered in accordance with each butterfly switch setting, and are intended to be active in DATA mode.

TABLE 4

BUTTERFLY SWITCH CONTROL MODES

ACKA	REQA	MODE
0	0	IDLE
0	1	ADDRESS
1	1	DATA
1	0	RELEASE

To release the network after use, all processor source ports clear their request lines, and the REQA falling edge propagates rightward through the network to the processor destination ports without altering butterfly switch settings. To maintain continuity in RELEASE mode, we must pass the initiative for network disconnection to the destination side of the network. Accordingly, resetting REQA from DATA mode causes the destination ports to clear ACKA, and the falling edge transition travels to the source processor side of the network, completing the return to IDLE mode.

Finally, it is worth noting that although all processors typically issue address requests to the network at about the same time, they are not required to do so. If, for example, the upper channel of a butterfly switch receives an address request while the lower channel is still in DATA mode, the new request will be forwarded if a STATE change (crossed or straight) is not needed. Otherwise, the butterfly switch will be considered "busy" until the lower channel's mode reverts to IDLE, and its connection is no longer needed. The implementation of the butterfly controller is described below.

4.3.2 Butterfly Controller Implementation

All butterfly switches must operate in accordance with the network control modes of Table 4. Each butterfly controller within the 4-node network is a self-contained state machine which operates synchronously within itself, but asynchronously with respect to other network nodes and switches. As shown in Figure 14, the controller employs an internal feedback loop, and consists of an oscillator, input and output registers and a combinatorial logic section. The logic of Figure 14 is currently implemented as a 2Kx8 bipolar PROM, which permits a full controller implementation using four chips. The controller's operation is described in Appendix B in terms of a set of operating rules and twelve resultant logical equations.

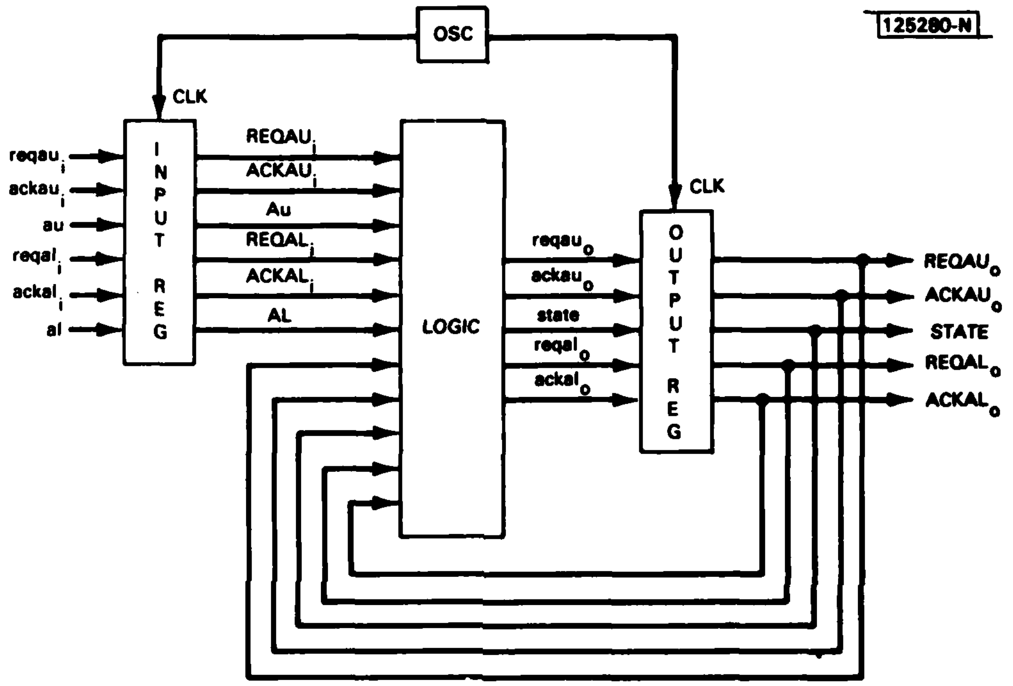


Fig. 14. Butterfly Controller Block Diagram

4.4 SPARING SWITCHES

As shown in the system block diagram of Figure 11, the DISP demonstration hardware includes sparing switches and a timing and fault manager with sparing switch interface. These resources are used (1) to attain a modest level of fault tolerance through the on-line substitution of a spare nodal processor when any one processor within a pool of five fails, and (2) to provide a means for issuing synchronizing triggers which will cause the four active nodal processors to begin their signal processing activities in unison.

Figure 15 shows the 2:1 multiplexer and 1:2 demultiplexer sparing switch arrangement, wherein any one faulty nodal processor may be excluded in accordance with the connection rules of Table 5. Implicit in Table 5 is the notion of more than one nodal processor being reassigned when a faulty processor is disconnected. This means that every processor must, for the one-spare case, have a "back-up" processor which contains the needed parameters, tables, etc. to operate in a reassigned role.

TABLE 5

NODAL PROCESSOR SPARING CONVENTIONS

Data Input (D)	Function	Sparing Enable (E_s)	Trigger Enable (E_t)	Sparing Output (S)	Spare Proc'r
0	Sparing	1	0	F	P_0
1		1	0	E	P_1
2	Switch	1	0	C	P_2
3		1	0	8	P_3
4	Select	1	0	0	P_4
5	Trigger	0	1	0	REQNET=1
6	Unused				
7	"				

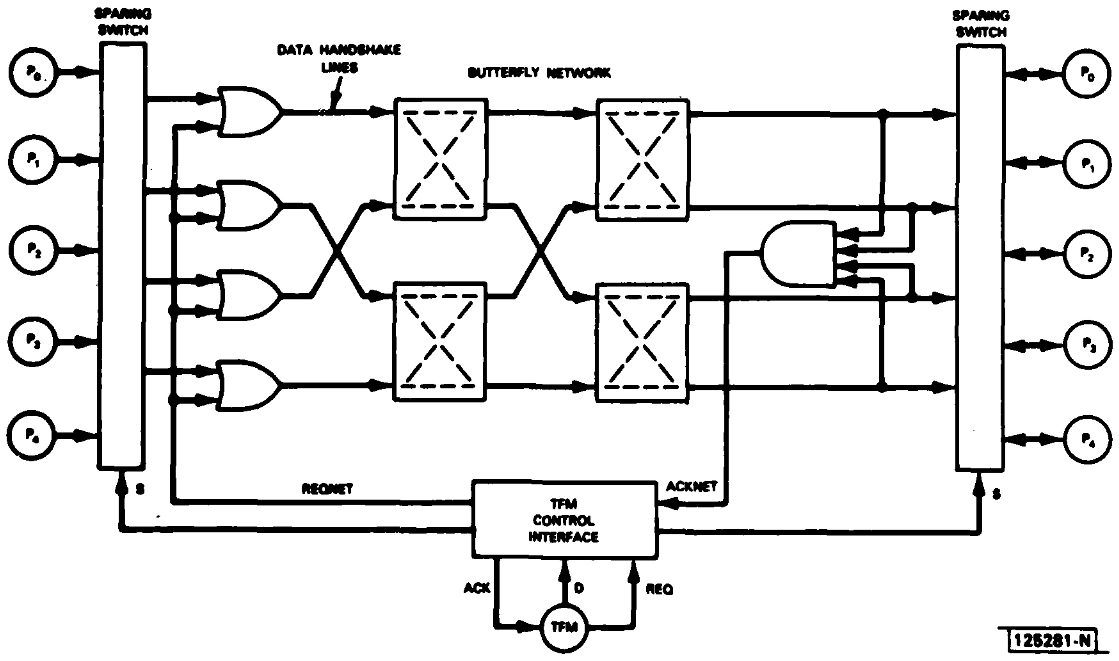


Fig. 15. Sparing Network for the DISP Test-Bed

4.5 TIMING AND FAULT MANAGER

The timing and fault manager (TFM) has two functions: (1) control of the sparing switches, and (2) providing top-level synchronization for the nodal processors. Like the nodal processors, the TFM was implemented using a VM01A microcomputer (Section 4.2). A simple interface was built to provide these control functions. (Figure 16). The interface is driven by one of the parallel ports of the TFM microcomputer.

As shown in Figure 16, a change in sparing switch setting begins by supplying the correct 3-bit D word to the interface, followed by a request (REQ). For all sparing codes, the enable trigger signal (E_t) is zero, and no request signal (REQNET) will enter the network. Instead, the local interface register is clocked by the TFM request (E_s=1) and the successful updating of the sparing output (S) together with the assertion of the request generates an acknowledge (ACK) which is returned to the TFM.

The TFM can also issue a common trigger signal to all NPs by means of the interface. Figure 15 shows in a general way the steering associated with data, data handshake, and butterfly network address handshake lines. Our discussion of triggering refers only to data handshake lines. Given that normal data requests emanating from the active nodes *and traversing the network are suppressed prior to triggering,* the TFM can inject a common request into each network channel which will be received simultaneously at the destination ports of all active nodes. This trigger function is independent of network configuration. Each node responds to the trigger by sending a data acknowledge which traverses the network, notifying the nodal source ports (via an "unsolicited acknowledge") that a trigger has been confirmed. The primary trigger acknowledge path, however, includes the four-input AND gate of Figure 15 where the *coincidence of* data acknowledges supplies the manager processor with a trigger acknowledge. The TFM interface is able to distinguish between data acknowledge signals arising from normal, distributed transfers across the butterfly network and data acknowledge signals in response to TFM triggers.

When the interface is used for triggering purposes, enabling levels E_t and E_s are complemented such that the interface register and comparator are inoperative, the TFM request causes REQNET to traverse the network, and the collective acknowledge ACKNET is received as a trigger acknowledge by the TFM. If the TFM interface is left in trigger mode while subsequent nodal processor parallel data transfers proceed, the resultant ACKNET signals can be monitored by the manager processor and serve as an indicator of epoch completion. However, the TFM need not know the details of network addressing or signal processing occurring within the nodes.

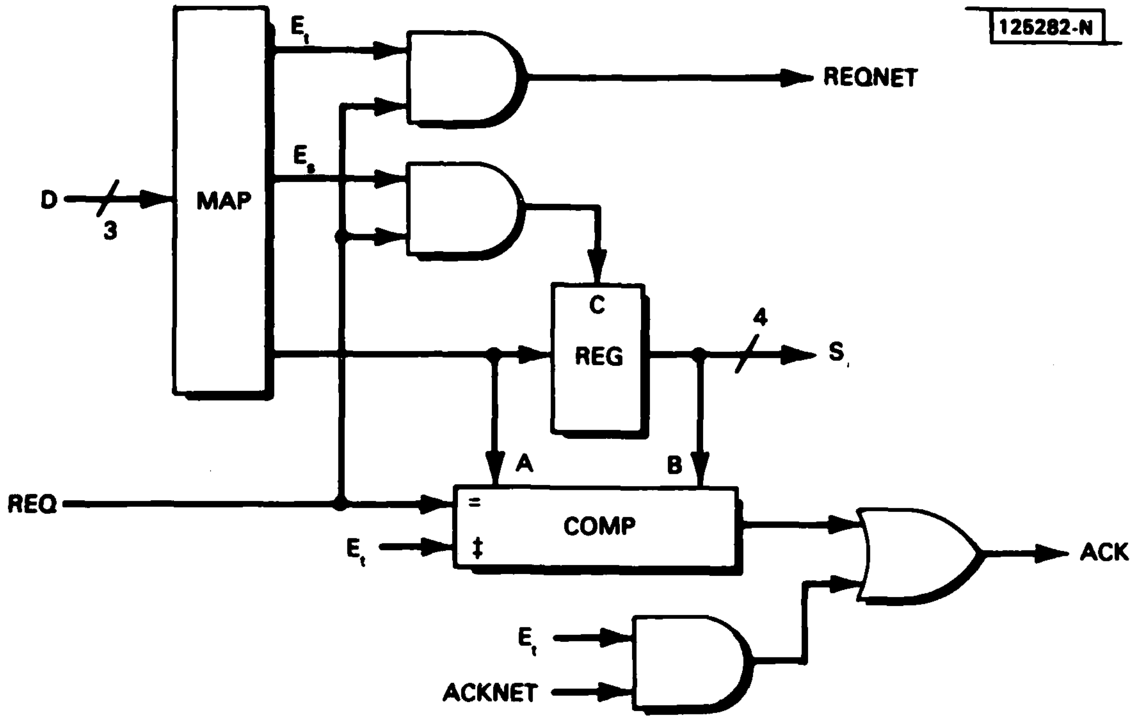


Fig. 16. TFM Control Interface

4.6 MULTI-PROCESSOR FFT (MPFFT)

4.6.1 Introduction

The initial goal of the DISP test-bed was to demonstrate the distributed computation of a one-dimensional FFT algorithm. This goal was motivated by the importance of FFT computations in projected applications of the DISP. Also, the distributed calculation of FFTs is an excellent way to exercise the various features of the DISP, i.e, calculations using the MC68000 microprocessor, use of interrupt capabilities and I/O ports, and inter-processor data transfers through the butterfly network. For simplicity, the initial MPFFT effort was limited to one-dimensional data sets.

In developing an MPFFT algorithm, our goal was to minimize the software development required. Thus, an algorithm was desired which is closely related to conventional single-processor FFT algorithms for which computer programs exist. A regular structure for data transfers was also desirable because this simplifies the portions of the program which control data flow through the butterfly network.

The MPFFT algorithm described in the following sections is a straightforward extension of a single-processor FFT (SPFFT) algorithm and thus meets all of these goals. Only minor coding modifications to a SPFFT program were needed to develop a subroutine which can be used to compute FFT's in either a single-processor or multi-processor mode. In addition, the data transfers have a regular structure which can be easily implemented in the program.

4.6.2 Partitioning the FFT Algorithm

There are many single-processor FFT algorithms which could be the basis for a MPFFT algorithm. To achieve a working algorithm in a reasonable time, some arbitrary restrictions were made. Specifically, it was decided to use a radix-2, in-place, decimation-in-time (DIT) algorithm with input data in bit-reversed order. Another motivation for this decision was provided by the existence of a working FFT program coded in MC68000 assembly language. Both the number of input data points and the number of processors were assumed to be integer powers of 2. A flowgraph for this algorithm applied to a 8-point data set was shown in Figure 6

A partitioned version of the algorithm for a 16-point FFT is shown in Figure 17. To aid in understanding the algorithm, the data points at each stage of the FFT calculation are labeled with the memory locations they would occupy in a single-processor version of the FFT. The exponents shown on Figure 17 are those which are generated by the SPFFT algorithm. For example, at the conclusion of the third butterfly stage in processor P_2, the data stored in locations 1 and 3 are

$$x_2^{(3)}(1) = x_2^{(2)}(1) + W_{16}^{2}\, x_2^{(2)}(3) = x^{(3)}(9)$$

$$x_2^{(3)}(3) = x_2^{(2)}(1) - W_{16}^{2}\, x_2^{(2)}(3) = x^{(3)}(13)$$

where the superscript gives the value of the stage variable, L, the subscript denotes the processor number and the argument specifies the memory locations. The variables $x^{(3)}(9)$ and $x^{(3)}(13)$ are the data values which would be found in a single-processor implementation in memory locations 9 and 13 at the completion of the third butterfly stage.

The first effect of partitioning we observe from Figure 17 is that there are two types of butterflies. The butterfly stages that are accomplished prior to any data transfers are called the *internal butterflies.* Those which are carried out after data transfers are called *external butterflies.* For a single-processor FFT of an N-point data set, $\log_2 N$ stages of butterflies are required. If the N data points are partitioned equally among P processors, each processor will have N/P points and the internal butterflies will consist of $\log_2(N/P)$ stages. Thus, the partitioned algorithm has $\log_2(N/P)$ internal butterfly stages and $\log_2(P)$ external butterfly stages.

Each external butterfly stage requires one block data transfer, so there are also $\log_2 P$ transfers. The pattern of these block data transfers is very regular. In particular, each transfer between two processors consists of an exchange of the lower half of the data in one processor with the upper half of the data in the other (see also Figure 7). To determine the destination processor in a transfer we use the cube_i functions defined by Siegel, et al [17]. The P processors are labeled P_0, P_1, ..., P_{P-1}. If we express each address as a binary number the bit pattern is

$$b_{j-1} \ldots b_1\, b_0$$

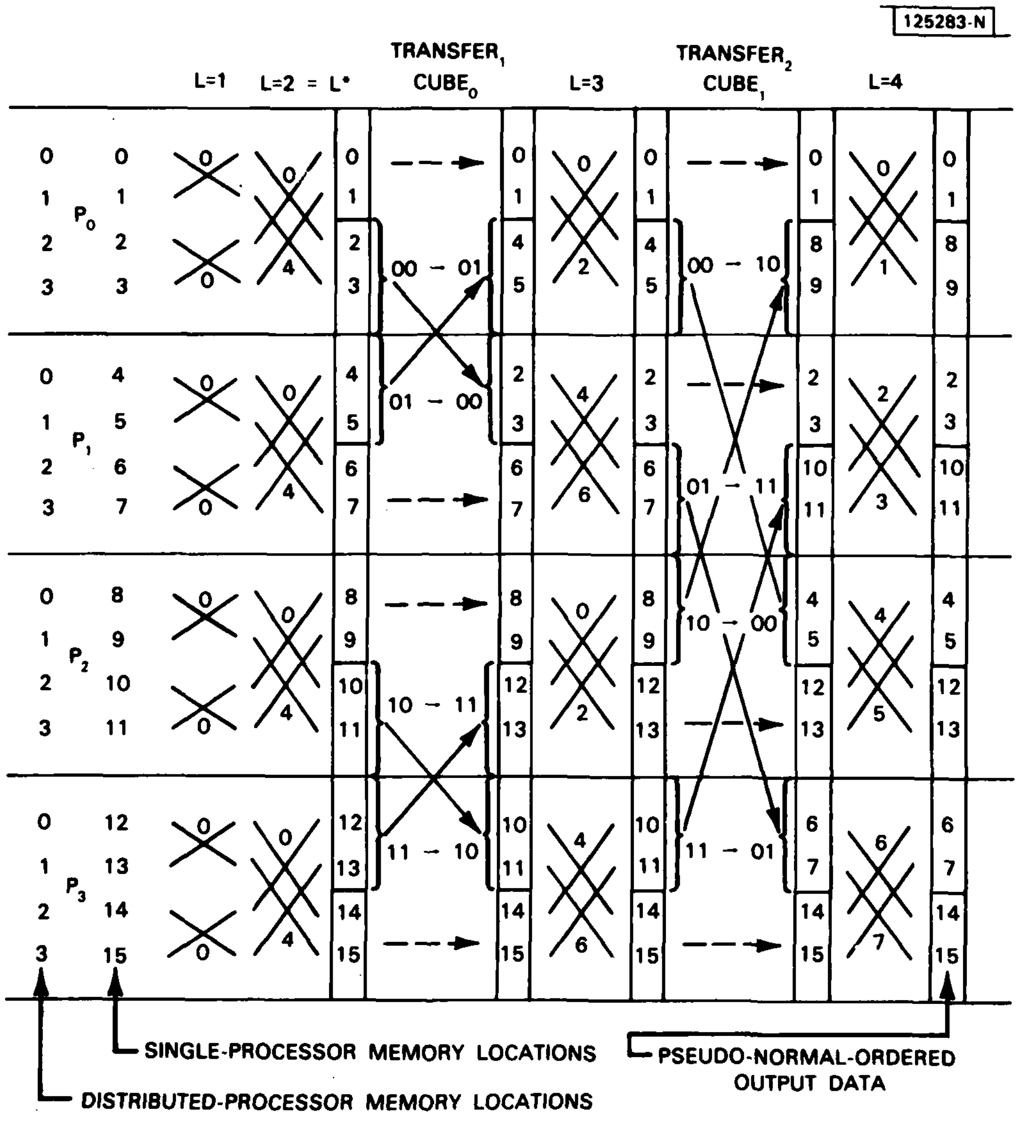


Fig. 17. Partitioned FFT Algorithm

where 2^j = P (notice the assumption that the number of processors is an integer power of 2). The cube_i function of this bit pattern is

$$b_{j-1}\ b_{j-2}\ \ldots\ b_{i+1}\ \overline{b_i}\ b_{i-1}\ \ldots\ b_0$$

that is, the ith bit is complemented. For example, if P=4, as in Figure 17, and we consider processor P_0 whose address is 00, the cube_0 function is 01 and the cube_1 function is 10. The destination processors in the first transfer are determined by use of the

cube_0 functions, those in second transfer by the cube_1, and so forth. Thus, in the first transfer P_0 transfers to P_1 and P_1 to P_0. Also, P_2 transfers to P_3 and P_3 to P_2. In general, the destinations in the ith transfer, $i=1,2,...,\log_2 P$, are determined by the $\mathrm{cube}_{(i-1)}$ functions.

It remains to specify whether a processor transfers the upper or lower half of its data.[3] This is accomplished by designating each processor as an "upper-half" or "lower-half" processor at each transfer stage. To determine in which half a processor belongs at the ith transfer stage, $i=1,2,...,\log_2 P$, we inspect the (i–1)st bit of the processor's binary address. If this bit is zero, the processor is an "upper-half" processor. It transfers the upper half of its data in the data exchange at this stage. A processor whose (i–1)st bit is 1, transfers the lower half of its data at the ith stage. For example, in the four-processor configuration of Figure 17 at the first transfer stage ($i=1$), the least significant bits of processors P_0 and P_2 are zero, hence these are upper-half processors; P_1 and P_3 are lower-half processors at this stage. At the second transfer stage ($i=2$), the b_1 bit of the processor's binary address is inspected. The result is that P_0 and P_1 are upper-half processors while P_2 and P_3 belong to the lower half.

As shown in Appendix A, the partitioning illustrated in Figure 17 has the very desirable property of leading to a simple algorithm. The algorithm requires that the input data be arranged in bit-reversed order *prior* to partitioning among the processors. Unfortunately, the output data is not in partitioned normal order, contrary to what occurs for a single-processor algorithm. The distributed processor algorithm produces the data in what we refer to as "pseudo-normal" order.

It should be pointed out that the transfers discussed here and illustrated for the four-processor, 16-point example of Figure 17 are not unique. Other patterns of transfers were investigated, but had the disadvantage of structures for which the general pattern could not be easily discerned and described analytically. Thus, these were discarded in favor of the regularly structured block transfers illustrated by Figure 17.

4.6.3 Determination of Butterfly Spacings and Computation of Weighting Factors

We are now ready to consider the spacing of points which are combined in the butterfly operations and the formulas which are necessary to compute the weighting factors. Our approach will be to show how the spacings and butterfly weights for a single-processor implementation can be modified to generate an MPFFT algorithm.

The spacing of points required in the FFT version used in the test-bed is particularly simple. For a single-processor algorithm as illustrated in Figure 6, the first butterfly stage requires data in adjacent memory locations. The second stage requires data in memory locations separated by two units; the third stage requires points separated by four units, and so forth. In general, at the Lth stage, the single-processor algorithm requires points separated by 2^{L-1} memory locations. The MPFFT algorithm proceeds in the same way (see Figure 17) until the point where data transfers must be made in order to continue the butterfly computations. Notice that beyond this point, the butterfly spacings are the same in each processor. If the transfer strategy

[3] Upper and lower refer to addresses in local memory, not to the relative positions in Figure 17.

described previously is used, the post-transfer butterflies in each processor require points separated in memory by $2^{(L^*-1)}$ units, where $L^*=\log_2(N/P)$. That is, the separation of data points required in all of the external butterfly stages is the same as for the last internal butterfly stage.

Next, let us consider the generation of the appropriate weighting factors. We will use

$$W_N^R(n) = e^{-j\,2\pi R/N} \tag{4-1}$$

to represent at a given stage the weighting factor needed in the nth butterfly. The butterflies in each processor are indexed starting with n=0 for the uppermost butterfly, n=1 for the next butterfly, and so on. The subscript N corresponds to the total number of points. The general expression for these weighting factors is

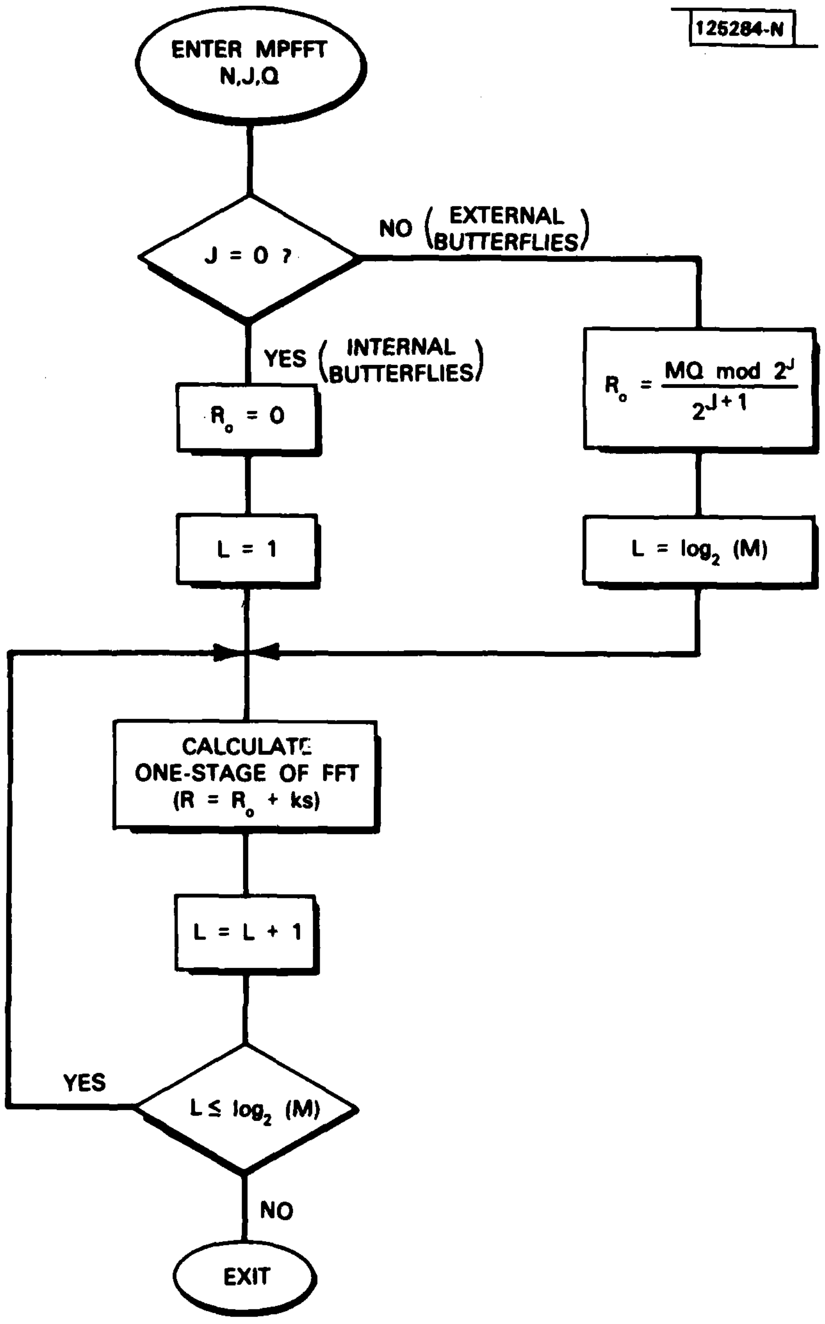
$$W_N^R(n) = W_N^{R_o+ks}\ (n) \tag{4-2}$$

In Appendix A, the expressions for the exponent $R=R_o+ks$ are derived.

4.6.4 The MPFFT Algorithm

The flowchart of the radix-2, in-place, decimation-in-time SPFFT algorithm is shown in Figure 18. This flowchart is identical to that of a conventional, single-processor FFT algorithm except for the addition of the upper-right-hand computation loop and the two new input variables, J and Q. The extra computation loop is invoked for external butterfly stages. It generates R_o, the initial value of the coefficient exponent, using the equations derived in Appendix A. The parameter J indicates whether internal (J=0) or external (J>0) butterflies are to be computed. When J>0, it is interpreted as the *external* stage number, and is related to the stage number, L, by L=J + log(N/P). Finally, Q is the *logical* processor address. Most of the computation is performed in the box labeled "Calculate One-Stage of FFT", which is identical to the corresponding portion of the SPFFT algorithm.

The MPFFT subroutine was coded in about 250 lines of MC68000 assembly language, of which 200 were taken from an existing SPFFT routine. The subroutine was extensively tested using the MC68000 simulator on the AMDAHL 470 computer. Simulation results using the MPFFT algorithm in a single-processor mode were compared with test results obtained using an APL subroutine on the AMDAHL 470. The results agreed to within the precision of the two machines. The SPFFT simulation results were then compared with MPFFT simulations. These two sets of results were identical for all of the test problems. Approximately a dozen test cases were run with a variety of numbers of points, different numbers of processors and different input data characteristics. After successful completion of these tests, the MPFFT subroutine was incorporated in the demonstration package and again tested successfully against the simulation results.



125284-N
ENTER MPFFT N,J,Q
J = 0 ?
NO (EXTERNAL BUTTERFLIES)
YES (INTERNAL BUTTERFLIES)
$R_o = 0$
$R_o = \frac{MQ \bmod 2^J}{2^{J+1}}$
L = 1
$L = \log_2 (M)$
CALCULATE ONE-STAGE OF FFT ($R = R_o + ks$)
L = L + 1
YES
$L \leq \log_2 (M)$
NO
EXIT

Fig. 18. Multi–Processor FFT Flowchart

The sequence of calls to subroutine MPFFT is shown in Figure 19. Notice that the subroutine TRANSFER is called to accomplish the necessary data exchanges among processors. The first call to MPFFT simply computes a conventional single-processor FFT with M=N/P points and J=0. An additional call to MPFFT is needed for each external butterfly stage, using the weights derived in Appendix A.

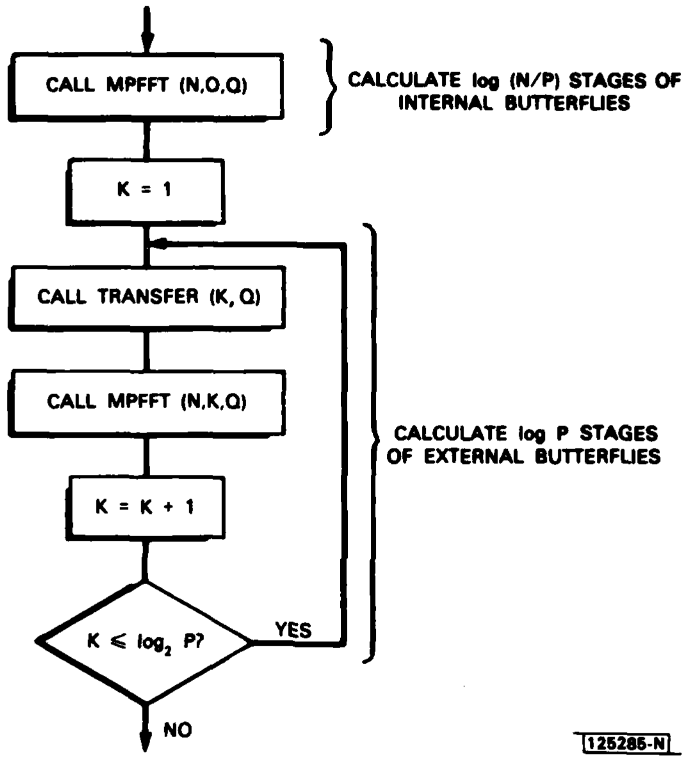

CALL MPFFT (N,O,Q)
CALCULATE log (N/P) STAGES OF INTERNAL BUTTERFLIES
K = 1
CALL TRANSFER (K, Q)
CALL MPFFT (N,K,Q)
CALCULATE log P STAGES OF EXTERNAL BUTTERFLIES
K = K + 1
K ≤ log₂ P?
YES
NO
125285-N

Fig. 19. Calling Sequence for Multi–processor FFT

4.6.5 Summary

The goals established for the MPFFT algorithm were satisfied. The SPFFT was readily modified to provide an MPFFT routine which can be used in either a single or multiple–processor mode. When used with a distributed system of processors, all processors execute the same program. Individual differences are accounted for through subroutine input parameters. We also observe that both internal (pre–transfer) and external (post–transfer) butterfly stages are computed by the same program. The $\log_2 N/P$ internal butterfly stages are computed by setting the flag/index J equal to zero before calling the MPFFT algorithm. Each external butterfly stage requires a separate call to the MPFFT subroutine with J= 1, 2, ..., $\log_2 P$.

4.7 MANAGER AND NODAL PROCESSOR SOFTWARE

4.7.1 Introduction

This section provides a top–level description of the software written for the timing and fault manager (TFM) and each of the nodal processors (NP) to carry–out the distributed calculation of a 1–D FFT on a four–node system. The description is given in terms of events appearing to an observer running the demonstration package. More detailed descriptions of the multi–processor time–line, the data transfer tables, the mailboxes, and the reconfiguration software are deferred to the Appendices.

Before leaving this introduction, it should be emphasized that very few changes would be needed to extend the applicability of the current software to an arbitrary number of nodes.[4] Furthermore, a very large part of this software could be reused directly to carry out other distributed calculations.

4.7.2 Operation of the Demonstration Package

A flowchart of the demonstration software package in shown in Figure 20. It shows the successive operations carried out cooperatively by the TFM, the five NPs (including one spare), and the Amdahl 470. Each box corresponds to a specific task in the demonstration. In the lower–right corner, the name(s) of the processor(s) mainly responsible for the task is given. The number in the lower–left corner corresponds to the following subsections.

The various tasks are now examined in detail and the corresponding computer outputs are shown in Figures 21–27. The composite outputs of Figures 21–25 have been obtained by appending two successive frames from the display terminal. The sequence of outputs shown in Figures 21–27 is automatically generated by the demonstration package, with minimum action required on the part of the user, such as typing "U" to upload, or "CR" (carriage return) to continue.

4.7.2.1 Turn on Selected Display Features (TFM)

The operator can choose among three levels of sophistication regarding the information displayed on the terminal. In the default setting, (a) messages are printed to allow the user to monitor the tasks being initiated or completed, (b) the input data and output results are plotted, and (c) the block diagrams illustrating the various processor interconnections are displayed. The block diagrams, or both the block diagrams and the plots can be omitted if desired, thereby leading to a faster and more realistic demonstration of the distributed FFT calculation and of the reconfiguration. The following discussion assumes use of the default option.

---

[4] Of course, this number must remain a power of 2.

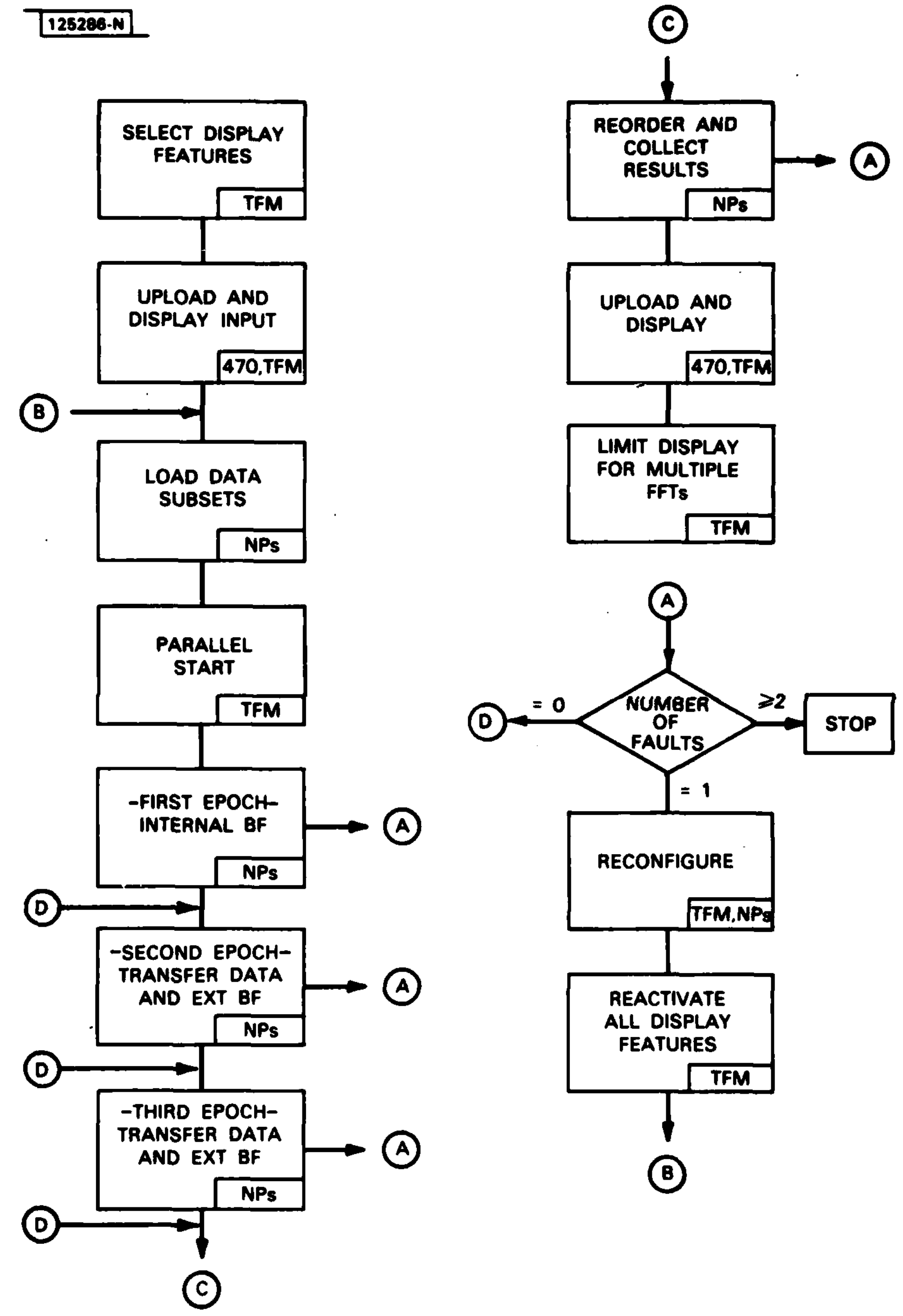


Fig. 20. Flowchart of Demonstration Package

4.7.2.2 Upload and Display Input Data (470 and TFM)

The data to be transformed are typically generated by the TFM, but can also be downloaded from the 470 if necessary. In either case, the TFM subsequently activates an executive routine in the 470, which then invites the user to initiate the upload by typing "U" (Figure 21). This transfers control to a TFM "upload" subroutine, which sends the data to be displayed to the 470, where the activated routine is waiting. After collecting the full data set, the 470 routine formats the data and generates, in "translucent" mode (see Appendix G), a display of the real part of the input data (Figure 21).

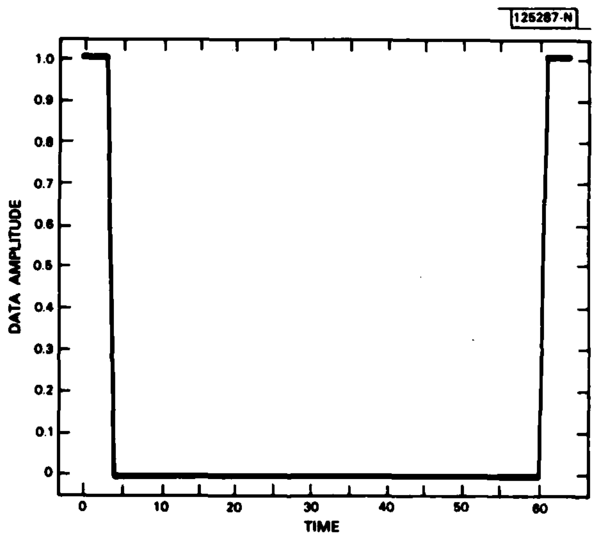


Fig. 21. Input Data Display

4.7.2.3 Load Data Partitions (NPs)

This step consists of making available to each active NP (excluding the spare) its share of the input data. With the present scheme, the bit-reversed data must be transferred to a portion of the memory address space that is common to all processors, namely the load-and-test memory of Figure 11. After being instructed to do so, each NP loads its portion of the data, as determined by its logical identity.[5] Proper completion of this step by all NPs causes the manager to print the message "data loading completed" on the screen as shown on the first line of Figure 22.

[5] Initially, the logical and physical identities are chosen to be identical (see Appendix C, Phase 1).

4.7.2.4 Parallel Start Nodal Processors (TFM)

This nearly-simultaneous start of the NPs is caused by the TFM, which broadcasts a "trigger" to all NPs (including the spare which ignores it). The message "parallel start issued" appears on the screen (Figure 22), indicating that the NPs have initiated their distributed signal processing operations.

4.7.2.5 First Epoch (NPs)

The first epoch consists of all butterfly calculations that can be done internally by each of the NPs (excluding the spare), i.e., without any exchange of information between processors. Upon completion of these "internal" butterflies, the message "first epoch completed" is generated by the TFM (Figure 22), thereby indicating that $\log_2(N/P)$ FFT stages have been computed. The TFM then sends a command to the 470 which responds by displaying the block diagram shown in Figure 22. It shows that the first epoch does not involve any transfer over the network, and also that the current spare is the NP with physical identity 4, i.e., P4. In order to proceed to the next epoch, the user only needs to type "CR", as suggested in the upper-right corner of the diagram.

4.7.2.6 Second Epoch (NPs)

The second epoch begins with concurrent transfers of data over the properly configured butterfly network, and is followed by internal calculations on the new combinations of data points available in each of the NPs. These butterfly calculations are called "external" to emphasize that, prior to the transfer, the corresponding FFT stage involves data present in two separate processors. The stage just computed is the $(\log(N/P)+1)$th one, but is often referred to as the first stage of "external" butterflies. Once this task has been completed by all NPs, the message "second epoch completed" appears on the screen (Figure 23), and is quickly followed by a block diagram showing the related processors' interconnections (Figure 23). A "CR" is all one needs to proceed to the next epoch.

4.7.2.7 Third Epoch (NPs)

This epoch, which is also the last one for a 4-node system, has the same structure as the previous one: it differs from it by the network configuration and the coefficients involved in the corresponding "external" butterflies. Figure 24 indicates the nature of the message and block diagram appearing on the screen upon completion of this task.

4.7.2.8 Reorder and Collect Results (NPs)

At this point, the results are available in "pseudo-normal" order in the processors P_0, P_1, P_2, and P_3 taken in that order . Because it is impossible to get direct

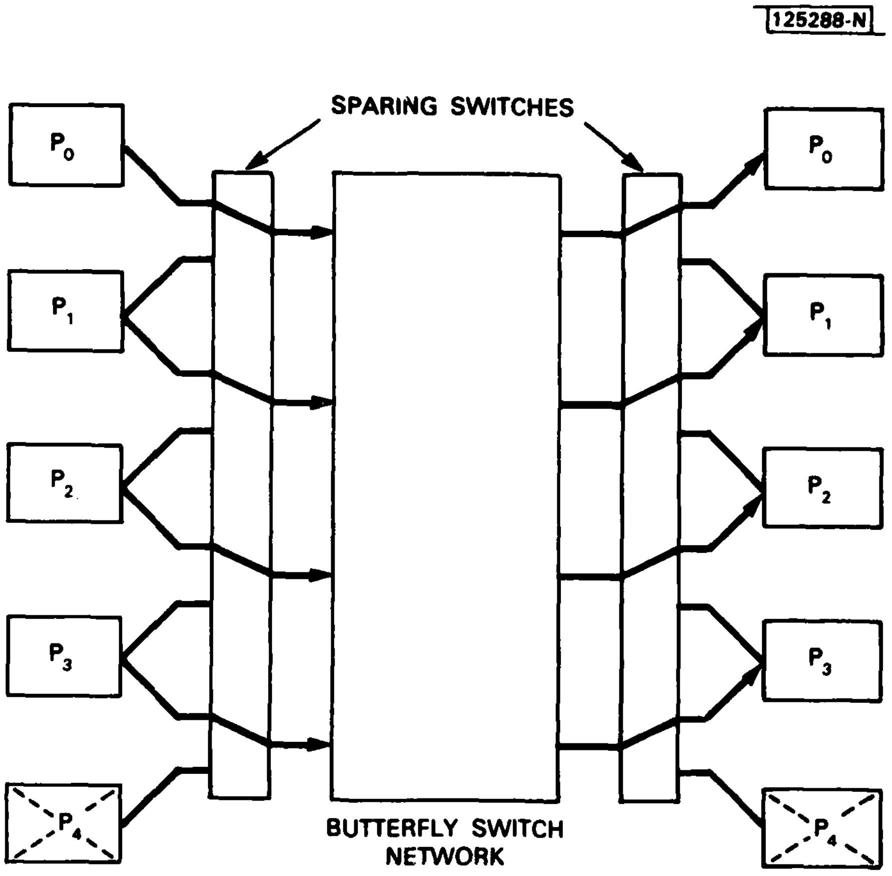


Fig. 22. First Epoch (Pre-transfer) Network Configuration

access to the full set of results from any single processor, the results are regrouped in the load-and-test memory, where they are available to all processors, in particular to the TFM and the 470. One could then have the TFM reorder the results if desired. However, it has been found that the transition from "pseudo-normal" to normal order could be achieved through two (log(P) in general) transfers through the network, namely, by successively using the third and second epoch network configurations, and doing the related transfers in opposite directions. While this technique would not be chosen for an operational system, it has been adopted in the experimental test-bed as a means of further exercising the network. The reordering by the NPs requires two successive configurations of the network, but the corresponding block diagrams are not shown. The whole reordering-and-transfer operation is completed when the message "result transfer completed" appears on the screen (Figure 25).

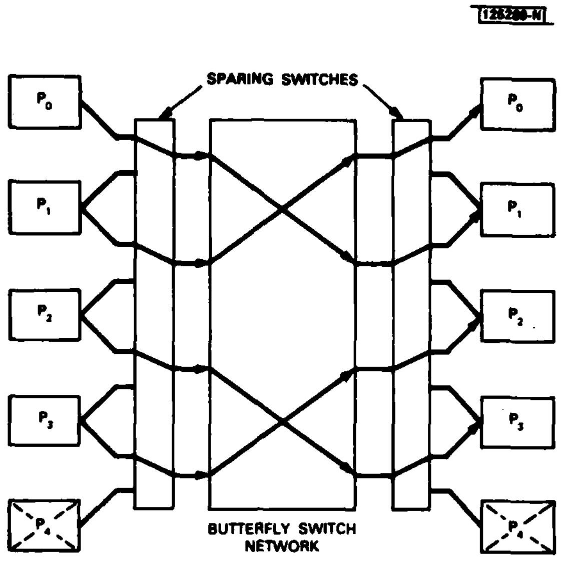


Fig. 23. Second Epoch (First Transfer) Network Configuration

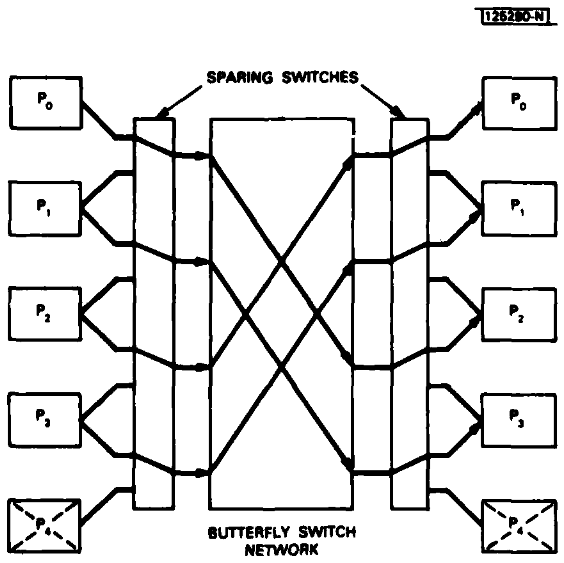


Fig. 24. Third Epoch (Second Transfer) Network Configuration

4.7.2.9 Upload and Display Results (470 and TFM)

The sequence of operations which leads from the array of results in load-and-test memory to the magnitude plot shown on the display (Figure 25) is identical to that involved in uploading the input data (Figure 21).

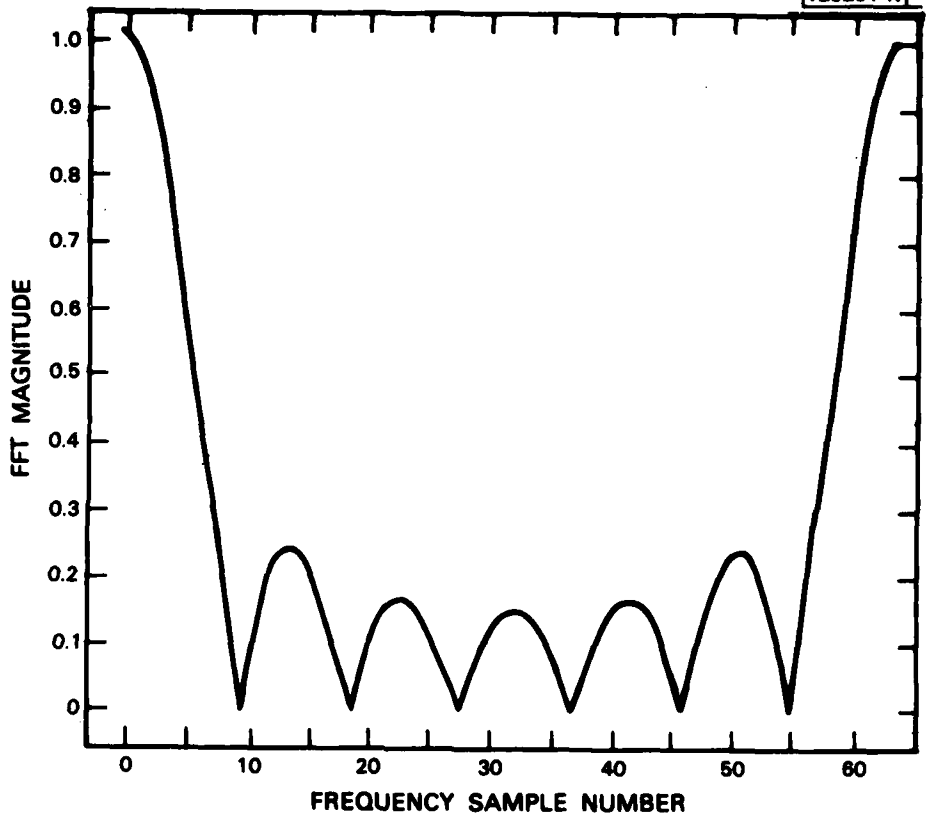


Fig. 25. Result of Distributed 1-D FFT

4.7.2.10 Select Display Features (TFM)

From this point on, new FFTs are repeatedl\ computed, and the output plots will be the only items displayed. This endless loop will be broken if faults are injected in one or more NPs via their individual fault-injection thumbwheels. This provides an artificial way of simulating true internal faults in the NPs and forcing the system to reconfigure. The thumbwheels are examined many times per FFT, as illustrated in Figure 20, to determine if a fault has been injected.

4.7.2.11 Reconfigure System (TFM and NPs)

If a single processor is found faulty, the system is automatically reconfigured. Although faults in up to 4 NPs could easily be handled by graceful degradation, whereby the number of active processors would be reduced to 2 or 1 as needed, this feature has not been implemented. The demonstration package simply stops upon detection of multiple faults.[6]

4.7.2.12 Reactivate Display Features (TFM)

Before resuming calculation of FFTs, the display features which were turned off earlier are reactivated, except for the input data plot. For example, if P_1 was designated as faulty, then Figure 23 would be modified as shown in Figure 26.

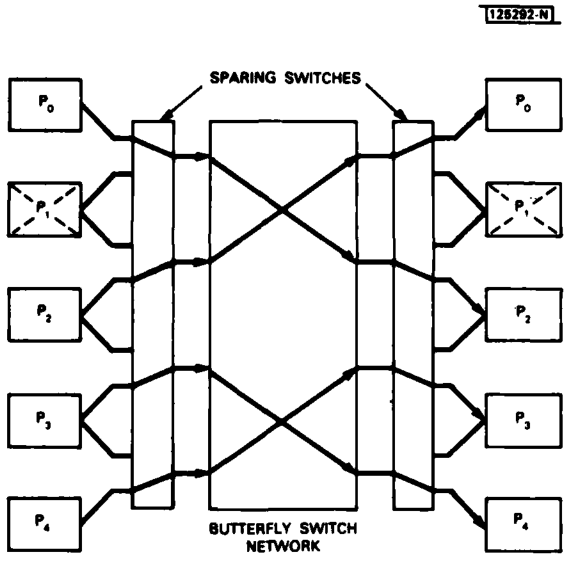
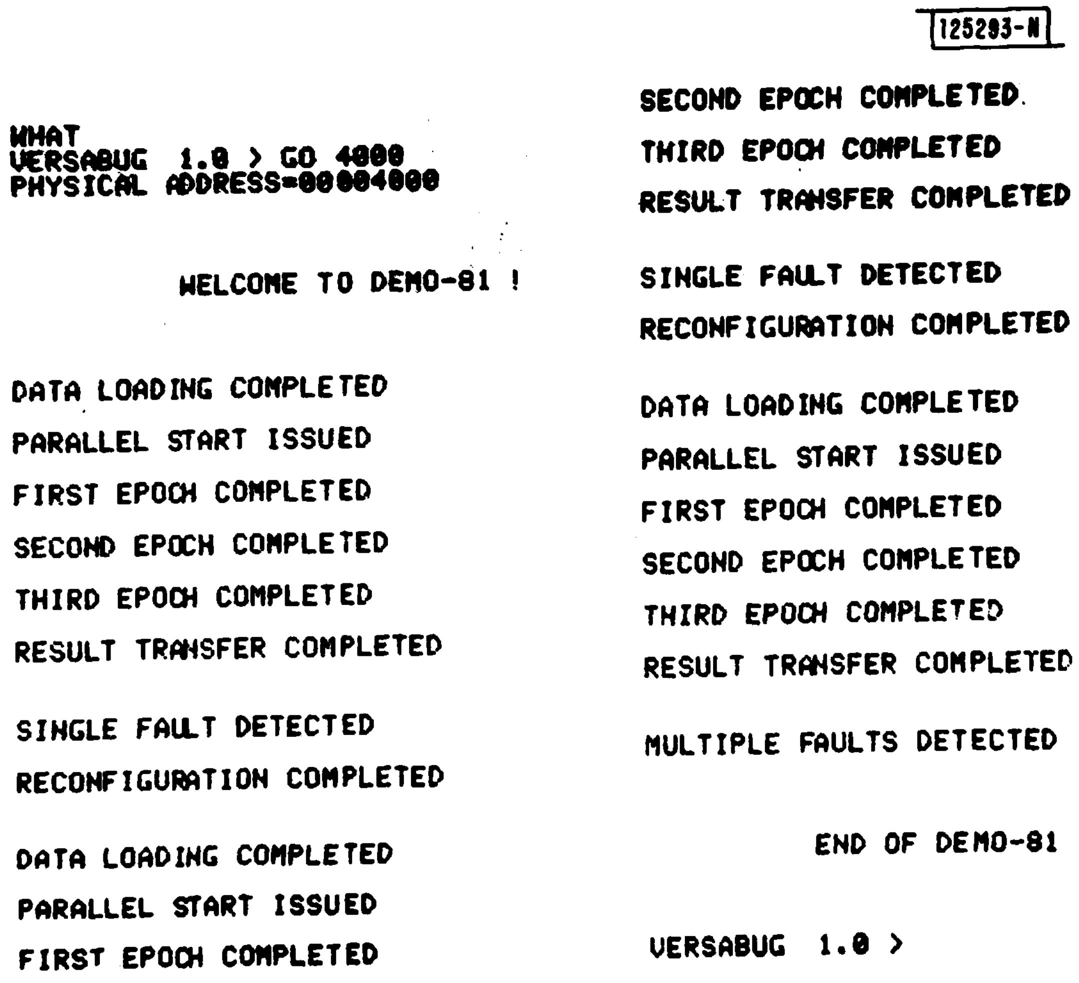


Fig. 26. Second Epoch (First Transfer) with P_1 "Faulty"

The newly declared-faulty processor becomes the new spare in the reconfiguration process and will remain so in all new calculations. If the thumbwheel of the faulty processor is brought back to its original position, the current spare is again ready to be called upon by the TFM upon injection of a new fault. Otherwise, if two thumbwheels are "simultaneously" found in their fault-injection positions, the system will not reconfigure. Figure 27 illustrates the reconfiguration mechanism through the

[6] Single fault refers to one or more faults in a single NP, while multiple faults refers to one or more faults in more than one NP.

console messages, i.e., with all plots and block diagrams suppressed. This corresponds to the case of two successive single-fault reconfigurations followed by a double-fault, which results in the termination of the demonstration package.


125293-N

```
WHAT
VERSABUG  1.0 > GO 4000
PHYSICAL ADDRESS=00004000

          WELCOME TO DEMO-81 !

DATA LOADING COMPLETED
PARALLEL START ISSUED
FIRST EPOCH COMPLETED
SECOND EPOCH COMPLETED
THIRD EPOCH COMPLETED
RESULT TRANSFER COMPLETED

SINGLE FAULT DETECTED
RECONFIGURATION COMPLETED

DATA LOADING COMPLETED
PARALLEL START ISSUED
FIRST EPOCH COMPLETED
SECOND EPOCH COMPLETED
THIRD EPOCH COMPLETED
RESULT TRANSFER COMPLETED

SINGLE FAULT DETECTED
RECONFIGURATION COMPLETED

DATA LOADING COMPLETED
PARALLEL START ISSUED
FIRST EPOCH COMPLETED
SECOND EPOCH COMPLETED
THIRD EPOCH COMPLETED
RESULT TRANSFER COMPLETED

MULTIPLE FAULTS DETECTED

          END OF DEMO-81

VERSABUG  1.0 >
```

Fig. 27. Reconfiguration Messages

4.8 SOFTWARE DEVELOPMENT TOOLS

4.8.1 Introduction

This section describes the software tools that were used to support software development for the DISP test–bed. The tools are: (1) Versabug, a firmware–resident monitor/debugger; (2) an absolute cross assembler resident on the Amdahl 470; and (3) a simulator for the MC68000, also resident on the 470. All three programs were supplied by Motorola.

In addition to these major tools, a number of utility routines were written to support and enhance the distributed FFT demonstration program. These routines are described in Appendix G.

4.8.2 Versabug (M68KVBUG(D1))

Versabug is a minimum function firmware–resident debugging package for the Versamodule Monoboard Micromputer. It is a limited capability monitor system stored in eight 2K x 8 EPROMs which physically reside on each Versamodule (VM01A) Versabug recognizes over three dozen commands, but in practice, only about half are used. Versabug was slightly modified to support the needs of the test–bed. These modifications are described in Appendix G.

When the VM01A is attached to a terminal, Versabug carries out a limited interactive dialogue with the user. When the same VM01A is also connected to the host 470 computer, Versabug can pass information between the user and host in "transparent mode".

A typical user–Versabug scenario starts with the user connecting to the host via the "TM" (transparent mode) command, downloading a previously assembled program from the host using the "LO" (load) command into the 32Kbyte on–board private RAM and then executing the program just loaded with the "GO" command.

With the program loaded, the user can interact with Versabug to set/remove breakpoints; display/modify memory; display/modify processor registers; initialize/test blocks of memory; trace through the program under test and finally upload blocks of memory to the host computer.

4.8.3 MC68000 Cross Macro Assembler (M68KXASM(D3))

The application programs for the DISP test–bed were all coded in assembly language using the cross assembler installed on the 470 computer. The procedure was to use the host editor to create or modify an assembly language source program file and finally to invoke the cross assembler.

The cross assembler is a standard two pass type, containing many useful assembler directives and featuring a macro facility with nesting up to three levels. It pro–

vides a printed listing containing the source language input, assembler object code, and error codes. It also generates an object code file in a format acceptable to the Versabug loader to permit downloading of the program from the host to the VM01A.

The principal shortcoming of the assembler is that it produces absolute object code. (Motorola now sells a relocatable cross-assembler plus linking loader which has been purchased and installed on the 470.) The absence of relocatable code and accompanying linkage editor forced the writing of long main programs rather than several small relocatable, linkable modules. Further, the origins of the principal modules had to be assigned absolute values known to the programs in absolute terms. This amounts to manual linking of the main modules.

4.8.4 MC68000 Simulator (M68K0SIM(D2))

A second non-resident programming tool that proved very valuable in debugging portions of the application software is the MC68000 simulator. It simulates the execution of an object program generated from a source program which had been assembled by the cross assembler. Since the simulator operates on the host machine in the time sharing mode, simultaneous multi-user operation of the simulator is possible.

The simulator is very effective in debugging computation-intensive code, but is of little help with input-output problems. The value of the simulator is that a user could debug all or parts of his program without using the Versamodule hardware.

The simulator command set is a subset of the Versabug command set. Included are load program under test, set/clear/display breakpoints, set and display memory/registers, run program and trace program.

4.9 EXAMPLES AND PERFORMANCE

4.9.1 Examples

Figures 28 and 29 illustrate the several stages in the calculation of the distributed FFT of a rectangular pulse and linear-FM chirp. Both examples contain 64 data points. In both cases, the first graph presents the input data set in normal order. Prior to distribution to the NPs, the *entire* array of data must be bit-reversed, and this is accomplished by the TFM. The bit-reversed array is shown in the second graph which clearly indicates that the first sixteen points in that array must be loaded into P_0, the next sixteen into P_1, and so on. The third graph shows the results in pseudo-normal order after computing all internal and external butterflies in parallel in each node. This unfamiliar ordering can be unravelled by the NPs through two additional transfers over the network. The result of this reordering is presented in the fourth graph.

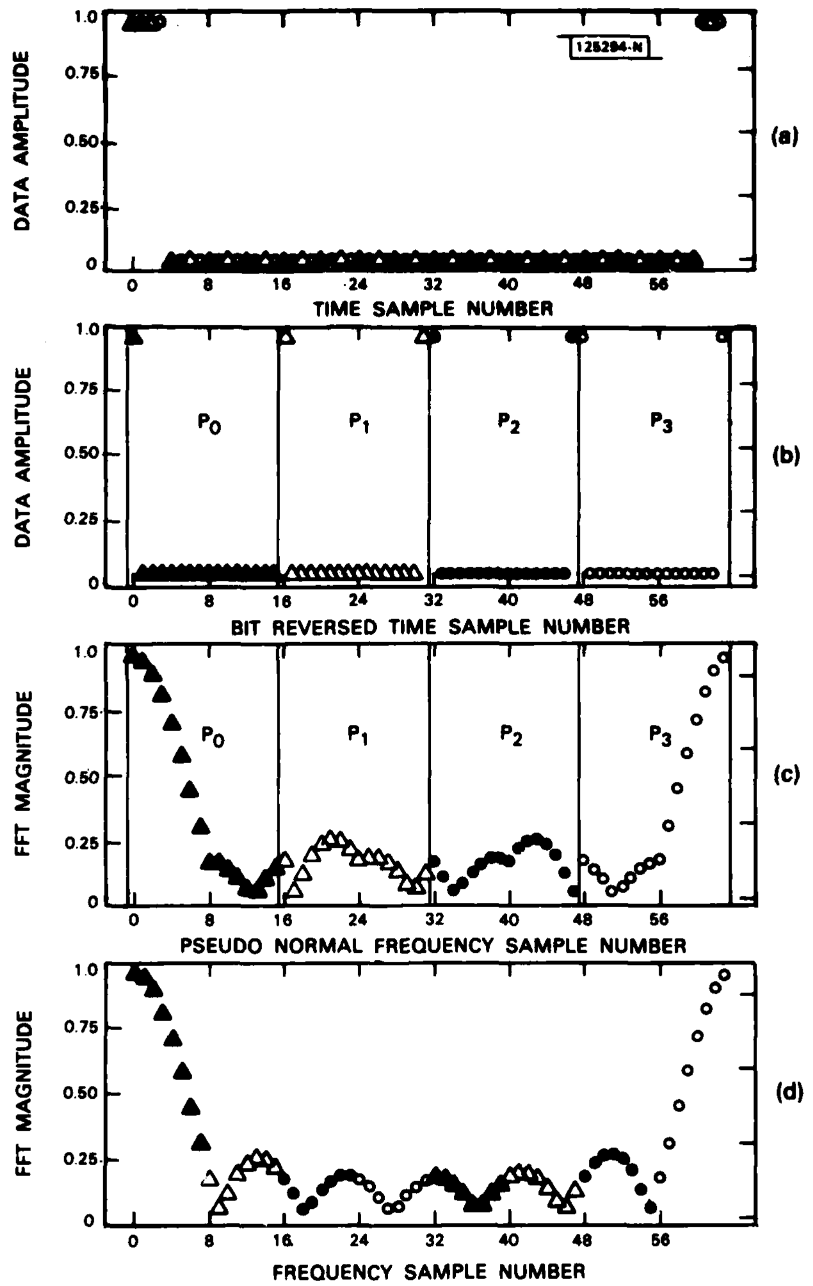


Fig. 28. Multi-Process r FFT of Rectangular Pulse

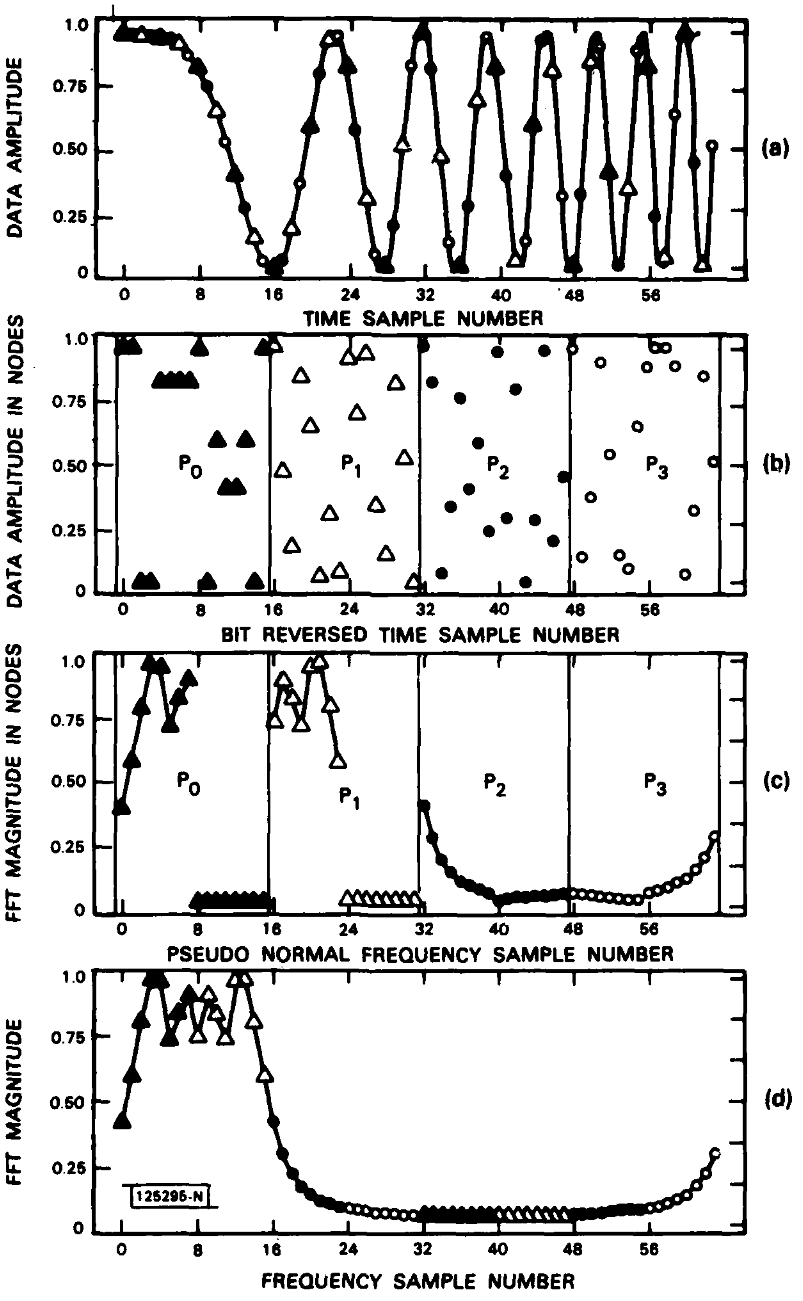


Fig. 29. Multi-Processor FFT of Linear FM Pulse

4.9.2 Execution Time

The critical sections of the distributed FFT are the butterfly (BFY) calculations, the FFT transfers, and the reordering transfers (optional). The rest can be regarded as overhead such as loading of data and precomputed results, and checking and transfer of final results. The execution times of the critical sections and of the overhead will be denoted by T(BFY), T(XFR), T(ORD), and T(OVH), respectively. These parameters can be determined indirectly by measuring the execution time of a number of distributed FFTs in four different situations, namely[7]

A. compute BFYs, do FFT transfers, do not reorder results
B. skip BFYs, do FFT transfers, do not reorder results
C. compute BFYs, do FFT transfers, reorder results
D. skip BFYs, do FFT transfers, reorder results

Denoting their respective execution times by T(A), T(B), T(C), and T(D), it is clear that

$$\begin{aligned}
T(A) &= T(BFY) + T(XFR) + 0 + T(OVH)\\
T(B) &= 0 + T(XFR) + 0 + T(OVH)\\
T(C) &= T(BFY) + T(XFR) + T(ORD) + T(OVH)\\
T(D) &= 0 + T(XFR) + T(ORD) + T(OVH).
\end{aligned}$$

These equations are approximate since, for example, it is not true that the overhead is absolutely identical in all cases. However, the differences are certainly very small. Because T(BFY) and T(ORD) can both be obtained from two different measurements, averages will be taken. Furthermore, with excellent approximation, T(XFR) is equal to T(ORD) since both involve the same types and numbers of transfers. Thus, in summary,

$$\begin{aligned}
T(TOT) &= T(C)\\
T(BFY) &= 0.5[T(A) - T(B) + T(C) - T(D)]\\
T(XFR) = T(ORD) &= 0.5[T(C) - T(A) + T(D) - T(B)]\\
T(OVH) &= T(C) - [T(BFY) + 2\,T(XFR)]
\end{aligned}$$

To determine the numbers T(A), ..., T(D) with a stop watch, the following procedure was used to eliminate any bias. In general T(X) was taken as

$$T(X) = T(X,2000) - T(X,1000) \text{ (result in msec)}$$

where T(X,J) represents the total measurement (in sec) corresponding to J FFTs. Table 6 shows the measurements obtained for a 64–point transform as well as the execution times derived from the above formulas.

For reference, the execution time of all butterfly calculations on a single processor is 39.65 msec. Note that it is tempting to compare T(BFY)=13.75 msec with 39.64 msec/4=9.91 msec. This comparison should, however, be done carefully because T(BFY) includes operations other than strict butterfly calculations, which cannot be

[7] These four cases can easily be generated by just changing the values of two parameters in the demonstration package.

TABLE 6

MEASURED EXECUTION TIMES

X	T(X,2000)	T(X,1000)	T(X)
A	98.6 sec	50.0 sec	48.6 msec
B	71.1 sec	36.1 sec	35.0 msec
C	165.5 sec	83.4 sec	82.1 msec
D	137.7 sec	69.5 sec	68.2 msec

T(BFY) =	13.75 msec
T(XFR) =	33.35 msec
T(ORD) =	33.35 msec
T(OVH) =	1.65 msec
T(TOT) =	82.10 msec

separated out with this simple method, because of the structure of the programs. The parameter T(BFY) actually involves: (1) true butterfly calculations, including the increase in complexity of the multiprocessor FFT program, (2) sending a message to the TFM, (3) looking once at a fault-injection thumbwheel. One should also point out that none of the above cases includes the "bit-reversing" operation, which is assumed to be carried out separately. Bit-reversal takes about 1.20 msec for 64 points on a single processor.

The timing figures given above clearly indicate that the network transfers are primarily responsible for the relatively poor performance of the distributed system. This limitation was recognized at the outset as the cost of building the test-bed with standard, single-board computers. An upgrade to the test-bed is planned which will include a DMA-driven, multi-ported memory to eliminate the data transfer overhead time. Software modifications which will overlap the transfers and calculations are also planned.

4.9.3 Lines of Code

One simple measure of the software effort is the number of assembly langaage *instructions*[8] used in the demonstration package. Table 7 summarizes these numbers for the Versabug modifications, the TFM program, the NP programs, and the multi-processor FFT.
Note that a large fraction of TFM instructions (307) are used to provide the display features such as messages, plots, and block diagrams.

[8] Comments are excluded.

TABLE 7

LINES OF ASSEMBLY LANGUAGE CODE

IN DISP DEMONSTRATION PACKAGE

Program	Number of Instructions
MOD to VBUG	128
TFM PGM	835
NP PGM	993
MPFFT	246
Total	2202

Chapter 5

SUMMARY

An architecture for performing distributed signal processing has been described. The emphasis on real–time signal processing simplifies the design because of the highly structured time–line and algorithms associated with such problems. It also permits the use of a simple, easily controlled butterfly network and results in managable, modular software.

The butterfly network provides simultaneous, conflict–free data transfers for multi–dimensional convolution (correlation) algorithms, and for a number of different *multi–dimensional* FFT algorithms.

A distributed signal processing test–bed has been constructed containing four active nodal processors, one spare, and the timing and fault manager. It has demonstrated and verified the concepts of (1) shared algorithm execution, (2) conflict–free data transfer, (3) distributed network control, (4) dynamic fault reconfiguration, and (5) identical software in all nodal processors.

Appendix A

DERIVATION OF MULTI-PROCESSOR FFT COEFFICIENTS

A.1 INTRODUCTION

Our starting point in deriving the multi-processor FFT coefficients (W_N^R) is the single-processor algorithm. In this case, an N-point transform has its butterflies indexed from n=0 to n=(N/2 −1). For the algorithm illustrated in Figure 6, the dependence of the weighting factors on the stage number, L, of butterfly calculations is shown in Table 8. In terms of the general expression given in Equation (A-1), it is readily seen that R_o=0 since the uppermost (n=0) butterfly at each stage has R=0, s=$N2^{-L}$ and k=n modulo (2^{L-1}), n=0,1,..., (N/2−1). The expression for k contains the modulo (2^{L-1}) factor because k is periodic with period 2^{L-1}.

$$W_N^R(n) = W_N^{R_o+ks}(n) \tag{A-1}$$

TABLE 8

SINGLE-PROCESSOR FFT COEFFICIENTS

Stage (L)	Range of R	Increment of R (s)	Period
$\log_2 N$	0→(N/2 −1)	1	N/2
$\log_2 N-1$	0→(N/2 −2)	2	N/4
$\log_2 N-2$	0→(N/2 −4)	4	N/8
.	.	.	.
.	.	.	.
L	0→(N/2 −$N2^{-L}$)	$N2^{-L}$	2^{L-1}
.	.	.	.
.	.	.	.
1	0→0	N/2	N/N=1

A.2 INTERNAL BUTTERFLY WEIGHTS

Next, let us consider the relationship of the internal butterflies to the single-processor algorithm. We wish to compute an N-point FFT by partitioning the computations among P processors, each having N/P = M points. We make the following assertion:

> The first $\log_2 M = \log_2(N/P) = \log_2 N - \log_2 P$ stages (the "internal" butterflies) of an N-point FFT can be computed by calling a "standard" single-processor FFT subroutine with number of points set equal to M = N/P.

The importance of the assertion is that we can execute the internal butterflies in the distributed-processor configuration by simply calling in each processor an identical SPFFT program with the number of points equal to N=(N/P).

To justify this assertion we first note that the SPFFT algorithm produces the correct spacings between data points which participate in a given butterfly. At the first stage, L=1, this spacing is 1 unit, for L=2 it is 2 units, and so forth until at the last stage, $L=L^*=\log_2 M$ and the spacing is (M/2)=(N/2P) units. The other requirement is that the correct weighting factors must be generated. First, we show that the weight patterns required for the individual processors are the same for the internal butterflies. From Table 8 we observe that the number of identical weight patterns, N_w, is

$$N_w = \frac{\text{Number of butterflies}}{\text{Period of weight pattern}} = N2^{-L} \qquad \text{(A-2)}$$

The minimum value of N_w for internal butterflies occurs at the maximum value of L which is at the last internal butterfly stage where

$$L = L^* = \log_2 N/P$$

So, for this stage, the number of identical weight patterns is

$$N_w = N/(N/P) = P$$

which is precisely the number of processors. Thus, each processor has one identical weight pattern at this stage. For $L \leq L^*$, each processor has its number of identical weight patterns equal to an integer power of 2. Therefore, for stages which can be computed prior to transfers all processors have identical weight patterns.

We now show that the complex exponential weighting factors generated by a standard SPFFT in-place radix-2 algorithm depend only on the stage variable, L, not on the number of points N. This means that the weighting factors produced by calling an identical SPFFT algorithm in each processor with number of points M=(N/P) will be the same as the first $\log_2 M$ stages of the same SPFFT algorithm executed for N points. Thus, to produce the internal butterflies we can call the same SPFFT algorithm in each processor with number of points equal to N/P.

The general expression for the weighting factors is

$$W_N^R = e^{-j\,2\pi R/N} = W^{R/N} \qquad \text{(A-3)}$$

where

$W = e^{-j2\pi}$.

From Table 8, the range of R is 0 to $N/2 - N2^{-L}$ in steps of $N2^{-L}$ and with a period of 2^{L-1}. Thus, there will be $(2^{L-1}-1)$ steps of R and the resulting values will be

$W_N^0, W_N^{1x2^{-L}N}, W_N^{2x2^{-L}N}, \ldots, W_N^{(2^{L-1}-1)x2^{-L}N}$

Using equation (A-3), this is equivalent to the steps

$W^0, W^{1x2^{-L}}, W^{2x2^{-L}}, W^{3x2^{-L}}, \ldots, W^{(2^{L-1}-1)2^{-L}}$

These steps depend only on the stage variable L.

Additional justification is provided by considering the $W_{N/P}^R = W_M^R$ values produced by calling the SPFFT program with the number of points equal to N/P. Again, using Table 8 with N replaced by N/P, R varies from 0 to $N/2P - (N/P)2^{-L}$ in steps of 2^{-L} N/P and with a period of 2^{L-1}. Thus, again there are $(2^{L-1}-1)$ steps and the $W_{N/P}^R = W_M^R$ values generated are:

$W^0_{N/P}$, $W_{N/P}^{1x2^{-L}N/P}$, $W_{N/P}^{2x2^{-L}N/P}$, $W_{N/P}^{3x2^{-L}N/P}$, $W_{N/P}^{(2^{L-1}-1)x2^{-L}N/P}$,

or, using Equation (A-3)

$W^0, W^{1x2^{-L}}, W^{2x2^{-L}}, W^{3x2^{-L}}, \ldots, W^{(2^{L-1}-1)x2^{-L}}$

which are the same as the values produced for the N-point FFT.

As an example, let's compare the weighting factors produced by calling the SPFFT algorithm with 64 points distributed among four processors with the weighting factors generated during the first four stages of the same SPFFT algorithm called with 64 points. Calling the SPFFT algorithm with the number of points equal to 64/4=16 we obtain:

L=1: $W_{16}^0, W_{16}^0, \ldots = W^0$

L=2: $W_{16}^0, W_{16}^4, W_{16}^0, \ldots = W^0, W^{1/4}, W^0, \ldots$

L=3: $W_{16}^0, W_{16}^2, W_{16}^4, W_{16}^6, W_{16}^0, \ldots = W^0, W^{1/8}, W^{1/4}, W^{3/8}, W^0, \ldots$

L=4: $W_{16}^0, W_{16}^1, W_{16}^2, \ldots, W_{16}^7 = W^0, W^{1/16}, W^{1/8}, W^{3/16}, \ldots, W^{7/16}$

Calling the same SPFFT program with number of points equal to 64, but considering only the first four stages (the internal butterflies), we obtain

L=1: $W_{64}^0, W_{64}^0, \ldots = W^0$

$L=2$: $W_{64}^{0}, W_{64}^{16}, W_{64}^{0}, \ldots = W^{0}, W^{1/4}, W^{0}, \ldots$

$L=3$: $W_{64}^{0}, W_{64}^{8}, W_{64}^{16}, W_{64}^{24}, W_{64}^{0}, \ldots = W^{0}, W^{1/8}, W^{1/4}, W^{3/8}, W^{0}, \ldots$

$L=4$: $W_{64}^{0}, W_{64}^{4}, W_{64}^{8}, W_{64}^{12}, W_{64}^{16}, W_{64}^{20}, W_{64}^{24}, W_{64}^{28}, W_{64}^{0}, \ldots$

$= W^{0}, W^{1/16}, W^{1/8}, W^{3/16}, \ldots, W^{7/16}, W^{0}, \ldots$

By comparing the W factors produced, we see they are identical.

To summarize, we have for the internal butterflies ($L \leq L^*$)

$W_N^R(n) = W_N^{R_o + ks}(n)$ where

$R_o=0$

$s=N/2^L$

$k=n \bmod(2^{L-1})$, $n=0,1,2,\ldots,(N/P-1)$

We now wish to develop the corresponding expressions for the external butterfly ($L>L^*$) weighting factors.

A.3 EXTERNAL BUTTERFLY WEIGHTS

By inspection of Figure 17 we observe that the number of butterflies which makes up a given weight pattern increases as the stage variable L increases. The number of identical weight patterns decreases correspondingly. Thus, for the external butterflies each weight pattern extends over more than one processor and the "starting value" R_o is not zero in each processor as was the case for the internal butterflies. This "starting value" of the exponent is periodic, however, in the linear array of processors. Finally, the increment in R, denoted by s, is given by the same formula in Table 8 which applies for the internal butterflies. Let us now make these observations quantitative.

First, let us consider the period of the starting value R_o. From Equation (A-2), the number of identical weight patterns, N_w, is $N2^{-L}$. For the external butterflies $L=L^*+1, L^*+2, \ldots, \log_2 N$, where $L^*=\log_2(N/P)=\log_2(M)$. It is easily shown that:

$N_w = P/2$ for $L = L^*+1$

and

$N_w = 1$ for $L=\log_2 N$.

Thus, the number of identical weight patterns satisfies

$N_w < P$, $P \geq 2$

so each weight pattern extends over more than one processor. The period of the weight patterns and hence the value of R_o is given by the number of processors per weight pattern, i.e.,

$$\frac{P}{N2^{-L}} = \frac{2^L P}{N} \qquad \text{(A-4)}$$

Thus, for each processor whose address equals an integer multiple of $2^L P/N$, R_o begins a new period.

It remains to determine the values of R_o within a period. This is accomplished by noting that within the first weight pattern, the value of R_o at stage L is equal to the butterfly weight increment at stage L times the number of butterfliers per processor times the processor address. Thus, within a period

$$R_o \;=\; N2^{-L} \times N/2P \times Q = \frac{2^{-L}N^2Q}{2P} = \frac{N^2Q}{2^{L+1}P} \qquad \text{(A-5)}$$

where Q is the processor address, Q=0, 1, 2, ..., P–1.

Taking into account the periodicity of R_o we have

$$R_o = \frac{N^2Q}{2^{L+1}P} \bmod \frac{2^L P}{N}$$

Thus, for the external butterflies, $L > L^* = \log_2 M$, we have

$$W_N{}^R(n) = W_N{}^{R_o + ks}(n)$$

where

$$R_o = \frac{N^2Q}{2^{L+1}P} \bmod \frac{2^L P}{N}$$

$$k = n\;,\quad n=0,\,1,\,2,\,\ldots,\,(N/2P-1)$$

$$s = N/2^L$$

Combining these expressions with those obtained for the internal butterflies yields

$$R_o = \begin{cases} 0, \text{ for } L \leq L^* \text{ ("internal" butterflies)} \\ \dfrac{N^2}{2^{L+1}P} Q \bmod \dfrac{2^L P}{N}, \text{ for } L > L^* \text{ ("external" butterflies)} \end{cases} \tag{A-6}$$

$$k = \begin{cases} n \bmod (2^{L-1}), \text{ for } L \leq L^* \text{ and } n=0, 1, 2, \ldots, (N/2P - 1) \\ n, \text{ for } L > L^* \text{ and } n=0, 1, 2, \ldots, (N/2P - 1) \end{cases}$$

$$s = N/2^L$$

where $L^* = \log_2 (N/P) = \log_2 M$

To summarize, we make the following observations about Equation (A–6)

1. It applies to a single processor (P=1) implementation as well as to the distributed processor case.

2. R_o is the "starting" value for R in each processor. For the internal butterflies ($L \leq L^*$), R_o is zero for all of the processors. For the external butterflies R_o depends on the processor number Q. This dependence is such that R_o is periodic in processor addresses with period $(2^L P/N) = 2^L/M$.

3. s is the "spacing" or increment in R. It depends on the stage number L and the number of points N, but not on the number of processors.

4. The modulo(2^{L-1}) factor in the expression for k reflects the periodicity of the internal butterfly weighting factors within a processor. For the external butterflies this factor is superfluous because for $L > \log_2(N/P)$, $2^{L-1} > (N/2P - 1)$ — each weight pattern extends over more than one processor. For the internal butterflies 2^{L-1} is the number of butterflies in each weight pattern. For example, in Figure 17 at stage L=2, a pattern consists of the weighting factors W^0_{16}, W^4_{16}. In general, with 2^{L-1} butterflies in each pattern and N/2P butterflies in each processor there are $(N/P)2^{-L}$ pattern repetitions in each processor for internal butterflies.

Next, we must relate the weighting factors for the external butterflies in Equation (A–6) to those produced by the SPFFT algorithm. From Figure 17 we observe that the memory spacing of the points which are paired together in external butterfly calculation stages is the same as for the L*th stage. This suggests the possibility of calling the standard SPFFT program once for each external stage with the stage variable fixed at the value L=L*. In fact this can be done, but modifications to the program are required to generate the appropriate weighting factors given by Equation (A–6). If we modify the SPFFT weighting factor expression to include a starting value x and step y,and call the subroutine with number of points equal to M, the program generates the weighting factors.

$$W_M^{x+ky} = W^{(x/M)+(ky/M)}, \quad k=0,\ 1,\ 2,\ \ldots,\ (M/2-1) \qquad \text{(A–7)}$$

The external butterfly weighting factors needed, as indicated by Equation (A–6), are

$$W_M^{R}(n) = W^{R/M}(n) = W^{(R_o/M)+(ks/M)}\ ,\ k = 0,\ 1,\ 2,\ \ldots,\ (M/2-1)$$

To establish the necessary correspondence we require

$$x = R_o = \frac{N^2}{2^{L+1}P} Q \bmod \frac{2^L P}{N} \qquad \text{(A–8)}$$

and

$$y = s = N2^{-L} \qquad \text{(A–9)}$$

Appendix B

BUTTERFLY SWITCH CONTROL EQUATIONS

B.1 INTRODUCTION

The controller portion of the full butterfly switch uses a subset of the variables shown in Figure 13. This subset explicitly includes input and output address requests and acknowledges. Additionally, one data bit from the upper and lower data buses (A_U and A_L) is picked off for control purposes, and the principal controller output (STATE) is generated for butterfly steering. All controller variables are named in Figure 14, where lower case signals are unregistered.

B.2 RULES AND EQUATIONS

The basic controller design philosophy requires that "next state" expressions be written for the five registered outputs shown in Figure 14, allowing for desired features and the effects of butterfly steering. In particular, the state and request outputs are based on an update/recirculate strategy, wherein each output defaults to the recirculate or "no change" condition whenever the criteria for update (i.e., change) are not fulfilled.

Several intermediate variables which simplify the derivation of the five output variables are included in the following discussion. These intermediate variables are $IDLE_U$ (upper channel idle), $IDLE_L$ (lower channel idle), IDLE (both channels idle), RS (recirculate STATE), NEWVAL (the new value of STATE upon update), RRU (recirculate request, upper), and RRL (recirculate request, lower).

It is essential to know when either or both butterfly channels (upper, lower) are active or idle, and this status information is not simply the result of decoding the mode variables of Table 4. Butterfly steering must also be taken into account. From the perspective of a requesting processor attached to the input side of a butterfly switch, the proper acknowledge signal is directly available, but the appropriate request signal depends on the prevailing straight/crossed condition. From Table 4, the IDLE condition is the logical NOR of ACKA and REQA. For the upper channel, $ACKA=ACKAU_o$, but

$$REQA=\overline{STATE}\cdot REQAU_o+STATE\cdot REQAL_o, \qquad \text{(B-1)}$$

resulting in

$$\overline{\mathrm{IDLE}_U} = \overline{\mathrm{STATE}}\cdot\mathrm{REQAU}_o + \mathrm{STATE}\cdot\mathrm{REQAL}_o + \mathrm{ACKAU}_o \qquad \text{(B–2)}$$

Similarly,

$$\overline{\mathrm{IDLE}_L} = \overline{\mathrm{STATE}}\cdot\mathrm{REQAL}_o + \mathrm{STATE}\cdot\mathrm{REQAU}_o + \mathrm{ACKAL}_o \qquad \text{(B–3)}$$

and for both channels collectively,

$$\mathrm{IDLE} = \mathrm{IDLE}_U\cdot\mathrm{IDLE}_L = \overline{\mathrm{REQAU}_o + \mathrm{ACKAU}_o + \mathrm{REQAL}_o + \mathrm{ACKAL}_o}. \qquad \text{(B–4)}$$

The rules for altering the STATE output include choice of update vs recirculate and, if update occurs, the specification of a new value. Updating changes STATE as a result of one or more requests. Recirculation maintains the present value of STATE, in accordance with the following rules:

Recirculate STATE if: 1) either or both channels active, or

2) both channels idle; no requests present, or

3) both channels idle; impending simultaneous conflicting requests.

Else update STATE.

Update and recirculate are complementary, and we define the recirculate STATE condition as:

$$\mathrm{RS} = \overline{\mathrm{IDLE}} + \mathrm{IDLE}\,[\overline{\mathrm{REQAU}_i}\cdot\overline{\mathrm{REQAL}_i} + \mathrm{REQAU}_i\cdot\mathrm{REQAL}_i \cdot(\mathrm{AU}+\mathrm{AL})] \qquad \text{(B–5)}$$

When updating STATE, a new value (NEWVAL) is calculated according to the following rules. NEWVAL is active if: (1) a single request with its accompanying address bit is present while the other channel is not requesting, or (2) two simultaneous requests occur which are not in conflict, in the presence of the address bit from at least one channel. Otherwise, NEWVAL is unused.

$$\mathrm{NEWVAL} = \mathrm{REQAU}_i\cdot\mathrm{REQAL}_i\cdot A_U + \mathrm{REQAU}_i\cdot\mathrm{REQAL}_i\cdot A_L + \mathrm{REQAU}_i\cdot\mathrm{REQAL}_i\cdot A_U\cdot A_L \qquad \text{(B–6)}$$

The complete description of the "next value" of STATE, then, is

$$\text{state} = \overline{\text{RS}}\cdot\text{NEWVAL} + \text{RS}\cdot\text{STATE} \qquad \text{(B-7)}$$

where, in terms of the nth internal controller clock, $\text{STATE}_{n+1} = \text{state}_n$.

The update/recirculate notion extends also to the controller request outputs, REQAU_0 and REQAL_0. Here the default condition is the updating of requests, and recirculation is invoked only when protection is sought against some forbidden event. The recirculate capability on requests offers protection against 1) preemptive requests on a channel which seek to disrupt traffic on an opposite busy channel, and 2) simultaneous conflicting requests on both channels for which the outcome may be uncertain. From the cases cited below, we may deduce the control conditions needed for upper and lower request recirculate commands, RRU and RRL.

First, consider single requests on the upper channel, with the lower channel active:

1) Condition: Impending conflict (cross request), STATE=0
Required Action: Update REQAL_0, recirculate $\text{REQAU}_0=0$

2) Condition: Impending conflict (straight request), STATE=1
Required Action: Update REQAU_0, recirculate $\text{REQAL}_0=0$

Next, consider single requests on the lower channel with the upper channel active:

3) Condition: Impending conflict (cross request), STATE=0
Required Action: Update REQAU_0, recirculate $\text{REQAL}_0=0$

4) Condition: Impending conflict (straight request), STATE=1
Required Action: Update REQAL_0, recirculate $\text{REQAU}_0=0$

Finally, consider the case of a fully idle butterfly switch:

5) Condition: Simultaneous conflicting requests
Required Action: Recirculate $\text{REQAU}_0=\text{REQAL}_0=0$

The recirculation capability associated with cases 1 through 4 prevents the active channel connection from being usurped by a requester. Such requests are honored as soon as the active channel becomes idle, and no retry is required. Therefore, preemptive requests are deferred, not denied. With resoect to conflict protection for simultaneous requests (case 5), the controller is essentially asked to do a "no-op", and not forward either the upper or lower channel request. One channel's request could have been honored, but there is no apparent advantage in doing so. Recall that simultaneous requests are honored if not in conflict.

The rules cited above can be coalesced to generate the logical expressions (RRU and RRL) for recirculation of $REQAU_o$ and $REQAL_o$. From cases 1, 4, and 5,

$$\begin{aligned} RRU = {} & REQAU_i \cdot IDLE_U \cdot \overline{IDLE_L} \cdot A_U \cdot \overline{STATE} + \\ & REQAL_i \cdot \overline{IDLE_U} \cdot IDLE_L \cdot \overline{A_L} \cdot STATE + \\ & REQAU_i \cdot REQAL_i \cdot IDLE \cdot (A_U \oplus A_L) \end{aligned} \qquad \text{(B–8)}$$

and from cases 2, 3, and 5,

$$\begin{aligned} RRL = {} & REQAU_i \cdot IDLE_U \cdot \overline{IDLE_L} \cdot \overline{A_U} \cdot STATE + \\ & REQAL_i \cdot \overline{IDLE_U} \cdot IDLE_L \cdot A_L \cdot \overline{STATE} + \\ & REQAU_i \cdot REQAL_i \cdot IDLE \cdot (A_U \oplus A_L) \end{aligned} \qquad \text{(B–9)}$$

The "next value" expressions for request outputs are of the form $REQAUo_{n+1} = reqauo_n$ and $REQALo_{n+1} = reqalo_n$ where n refers to internal controller clock cycles. Recalling that update and recirculate operations are complementary, and that updated requests are subject to butterfly steering, we have

$$requ_o = \overline{RRU} \cdot (state \cdot REQAL_i + \overline{state} \cdot REQAU_i) + RRU \cdot REQAU_o \qquad \text{(B–10)}$$

and

$$reql_o = \overline{RRL} \cdot (state \cdot REQAU_i + \overline{state} \cdot REQAL_i) + RRL \cdot REQAL_o \qquad \text{(B–11)}$$

The final two equations in the controller set are expressions for steered acknowledge outputs, i.e.,

$$acku_o = state \cdot ACKAL_i + \overline{state} \cdot ACKAU_i \qquad \text{(B–12)}$$

and

$$ackl_o = state \cdot ACKAU_i + \overline{state} \cdot ACKAL_i \qquad \text{(B–13)}$$

A Fortran program entitled NETROM generated the needed hexadecimal data file for electrically programming the 2Kx8 butterfly controller PROM. NETROM forms a two–dimensional input array (2048 addresses by 11 bits) and performs the computation embodied in equations (B–1) through (B–12) using Fortran logical variables. The resultant two–dimensional output array, after logical–to–integer conversion, provides the controller PROM table. A sixth output, $\overline{\text{state}}$, was added for hardware fanout purposes.

Appendix C

MULTI–PROCESSOR TIME–LINE

C.1 INTRODUCTION

The discussion presented in Section 4.7 provides the reader with enough information to interpret the visual outputs of the demonstration package. However, that discussion does not address the intricate timing of the events taking place in the various processors. A multi–processor time–line provides this extra information.

The multi–processor time–line was used in the initial design of the TFM and NP programs. This software design strategy is fairly general and well adapted to proof–of–concept experiments in a research environment. The six–phase structure described below is a natural candidate for any distributed task. Discussing details below the level of the time–line is beyond the scope of this report. The interested reader is referred to a collection of detailed flowcharts [25] or to the actual assembly code listings .

Carrying out the calculation of an FFT in a distributed way can be done with *four* basic software components, one of them being needed only for cold starts (i.e., immediately after power–up). A multitude of other support programs are, of course, required to (a) create the various object modules which are burned into EPROMs or eventually downloaded into the distributed system prior to execution, and (b) to help on the 470 side in uploading data and producing all graphical outputs. These programs are described in Appendix G.

The four software components of the demonstration package are listed below.

1. *System Clear program (SYSCLR PGM):* This memory initialization program is executed in the TFM prior to actual DISP operations, but this is only required after power–on.

2. *Timing and Fault Manager program (TFM PGM):* This program runs in the TFM to supervise the DISP operations carried out in parallel in the NPs.

3. *Nodal Processor program (NP PGM):* One copy of this program runs in all NPs.

4. *Modified Versabug program (MOD VBUG):* One copy of this program runs in all processors, i.e., in the TFM and all NPs.

When running the demonstration package, each of the six processor boards can be in either of two modes: (1) *Modified Versabug mode,* in which MOD VBUG is running, or (2) *Application Program mode,* in which TFM PGM or NP PGM is running.

Thus, there are four different programs operating among six different processors, not including the graphics–support programs and executives residing in the 470. The following sections describe the transition mechanisms from one mode to the other within each processor, as well as the communication and synchronization mechanisms among programs running in separate processors during the six–phase distributed signal processing sequence mentioned above.

C.2 EXECUTION PHASES

C.2.1 Phase 1: Start–up

From a programming viewpoint, all the processor boards are absolutely identical.[9] Thumbwheel switches located on the system front panel and connected to a parallel port on each board are used to provide each processor with its *physical* identity (Figure 30), which is dictated by its actual connections to the butterfly network through the sparing switches. Initially, the processor boards will behave on the basis of that physical identity. If the thumbwheels are used to artificially inject faults in the NPs, new *logical* identities will be assigned by the TFM through the communication mailboxes. The processor boards can thus be thought of as always acting according to their current logical identities provided the initial logical identities are taken to be equal to the physical identities. With the initial identities properly dialed on the thumbwheels, the power can be turned on and the whole system reset through the "system controller" reset button.

It is important, at this point, to mention that the critical "look–for–letter" bit (LFL bit), which is also part of the information available on the thumbwheel–connected parallel ports (Figure 30) is currently reset. Furthermore, all boards are now in Modified Versabug mode and quickly reach the Versabug modification (Appendix G), whose gross structure is shown in Figure 31. Since the LFL bit is reset, all boards bypass the Main Versabug Patch, which is described as part of phase 4. Since the modification to Versabug is inserted in the command input loop, the situation at the end of the start–up phase is as follows:

1. The TFM continuously monitors the terminal connected to it for new commands
2. Each NP continuously monitors the serial port where a terminal could be connected, as well as the LFL bit for a change of state. Under normal circumstances, no terminals are connected to the NPs, and thus no command is really expected. For debugging purposes, however, one has the option of switching the TFM terminal to any NP, or even connecting separate terminals to any number of NPs as needed.

---

[9] From a hardware viewpoint, the microcomputers generally differ through a few "jumper" settings, namely the ones related to the "system controller" function and the Versabus–access priority levels.

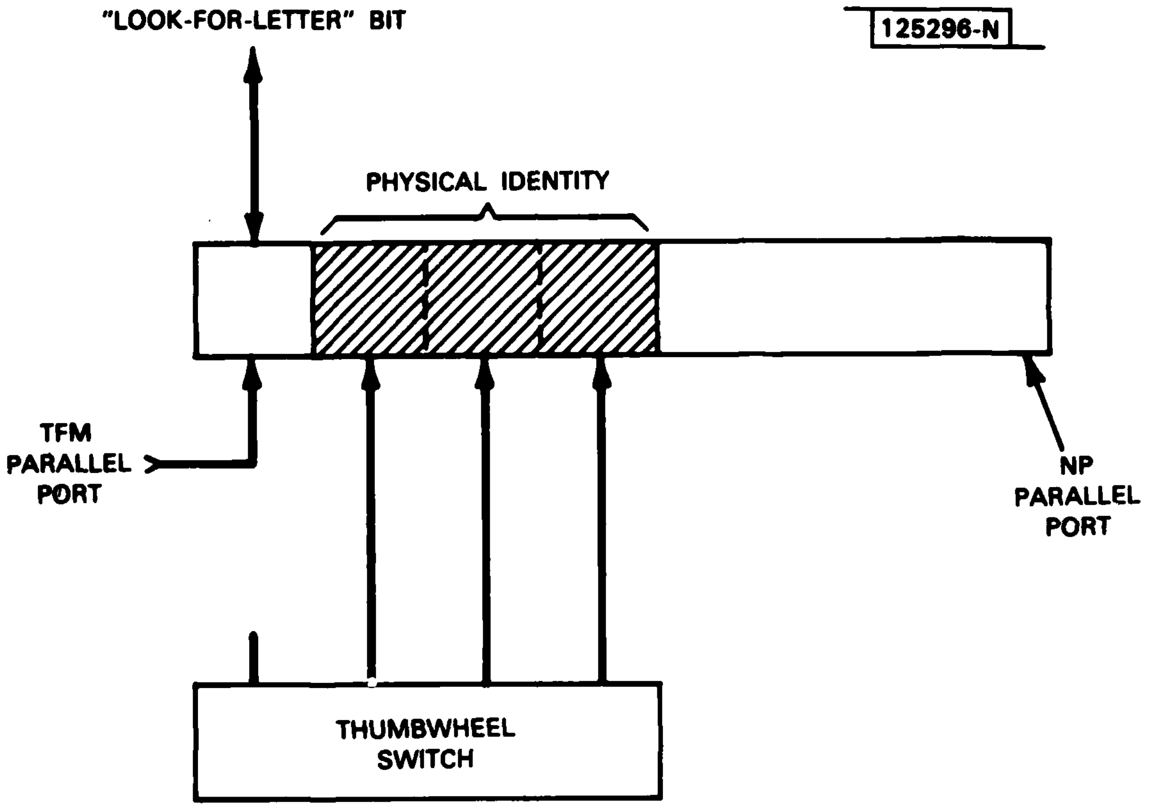


Fig. 30. Thumbwheel Switch Inputs

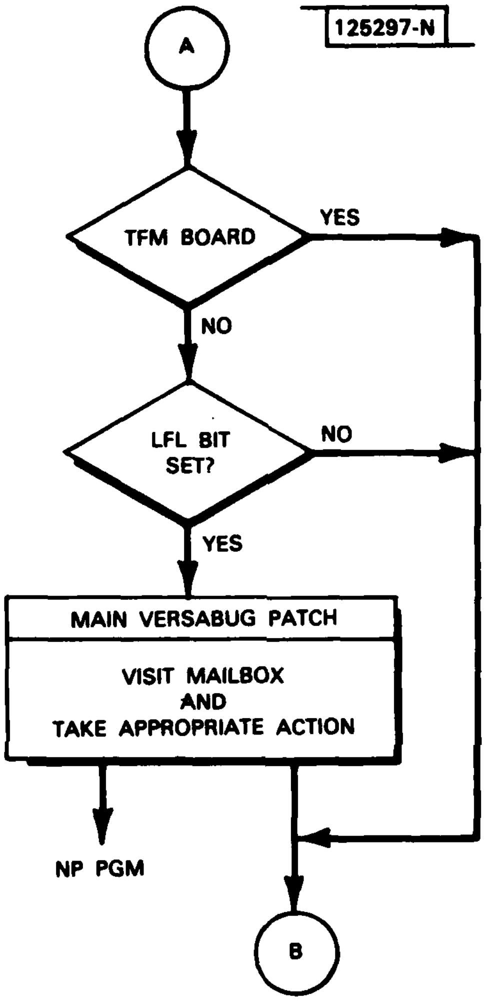


Fig. 31. Top–level Flowchart of Versabug Modification

C.2.2 Phase 2: Memory Initialization

The memory initialization phase is required to avoid the parity errors which occur when memory locations are first read.

C.2.2.1 TFM

The MOD VBUG loop described above is first broken by the DOWNLOAD command which loads the SYSCLR PGM in private TFM memory. It is broken a second time by a RUN command which forces the TFM to temporarily leave the Modified Versabug mode to run SYSCLR.

The SYSCLR PGM initializes the mailboxes in the load-and-test memory to avoid parity errors when the NPs start looking at their mailboxes after discovering that the LFL bit is set. Clearing the rest of the load-and-test memory is not really necessary, but this is nevertheless done to bring it into a known initial state. This program also initializes the TFM private memory because a later operation involves a transfer to load-and-test memory of a large block of code, not all of which would otherwise be initialized.

With the completion of the SYSCLR PGM, the TFM returns to the Modified Versabug mode, waiting for subsequent commands to be entered.

C.2.2.2 NPs

The NPs operation has not changed since phase 1. They are essentially waiting for the TFM to set the LFL bit, and this is done during phase 3.

C.2.3 Phase 3: Downloading and Preliminary TFM Operations

During this phase, the user downloads all the required object code into the TFM memory address space (i.e., private and load-and-test memory) and starts TFM PGM. After carrying out a number of preliminary tasks, this program transfers control to the Main Versabug Patch in all nodes. Note that NP PGM is not activated until the next phase.

C.2.3.1 TFM

The MOD VBUG loop is again broken to DOWNLOAD in a single operation the various modules.[10] Depending upon their starting address, these modules will be directed to either private TFM memory or to the load-and-test memory. In fact, the only module to go directly to load-and-test memory is the one containing the transfer tables (Appendix 32). These tables are of no use to the TFM and will be picked up later by the NPs as needed. The private memory naturally receives the TFM PGM but also, somewhat unexpectedly, the NP PGM as well, for the simple reason that a program module must be downloaded in the part of the addressing space where it will ultimately run. Since all boards have the same addressing spaces, it is natural to temporarily store NP PGM in the TFM private memory, provided it does not overlap with any other module. The multi-processor FFT module (MPFFT) is used both by the TFM (to precompute the results) and by all NPs, but is not, strictly speaking, a part of TFM PGM or NP PGM. In spite of this, it is conceptually useful to treat MPFFT as being an integral part of both programs.

The MOD VBUG loop is broken once more to start TFM PGM, thereby leaving the Modified Versabug mode (in the TFM only) until the end of the demonstration program. Control has thus effectively been transferred to the Application Program

[10] The downloading time of the whole demonstration package is limited by the 2400 baud link to the 470 and takes about 4 minutes.

mode (in the TFM only) which carries out the following preliminary operations.

1. Initialize the mailboxes with a properly selected code which will place the NPs in a hold state until further notice, even after the LFL bit becomes set.

2. Set the LFL bit, thereby allowing all NPs to safely access their mailboxes for the first time.

3. Transfer the NP PGM from its temporary location in private TFM memory to load–and–test memory (this includes MPFFT). In contrast with the initial download, this transfer can be done to an arbitrary address, by moving the NP PGM, one word at a time, under TFM PGM control. Moreover, the earlier initialization of the TFM private memory guarantees a transfer without spurious parity errors, without having to worry about the "gaps" generated by the "DS" (define storage) assembler directives which reserve memory locations but do not put anything into them.

4. Generate the data set to be transformed, typically a symmetrical pulse of selectable width[11] and compute its conventional (i.e., single–processor) FFT.

5. Transfer the data and the precomputed results to load–and–test memory, where, together with the transfer tables and NP PGM, they are accessible to all NPs.

6. Set–up the hardware sparing switches to provide the default system configuration in which the NP logical identities are equal to the physical identities.

7. Place a first letter in each of the NP mailboxes, which will force them to carry out a sequence of tasks in preparation to the actual distributed calculations.

C.2.3.2 NPs

Up to the very last TFM operation, the NPs are still in Modified Versabug mode. While they initially bypassed the Main Versabug Patch, they start going through it upon detection of a state change in the LFL bit. Even so, they are still being held in the same mode because of the absence of an appropriate letter from the TFM. The end of phase 3 corresponds with the reception of the first letter by each of the NPs.

It should be noted that in the usual case where FFTs are computed more than once[12] all the operations described in this and the previous phases for the TFM need only be done once, prior to the first FFT calculation.

---

[11] As announced earlier, an external data set can be used if one wishes. It must be downloaded separately in private TFM memory before running TFM PGM.

[12] This would be done to emulate a real system and/or demonstrate reconfigurability.

C.2.4 Phase 4: Loading and Preliminary NP Operations

This phase encompasses all the preliminary operations carried–out by the NPs before actually transferring control to their personal copies of the NP PGM, thereby leaving the Modified Versabug mode until the end of the demonstration.

C.2.4.1 TFM

The TFM PGM is running in a tight loop waiting for the first letters generated by each of the NPs to announce the completion of the various tasks described in this and part of phase 5.

C.2.4.2 NPs

Each NP is busy executing the tasks specified in the Main Versabug Patch, namely: (1) acknowledge the first letter just received from the TFM, (2) initialize the NP private memory to restore it to a known state, (3) load a copy of the NP PGM from load–and–test memory, (4) transfer control to NP PGM, thereby entering the Application Program mode.

All of the NP operations mentioned up to this point (i.e., in this and the previous phases) are not repeated after the first FFT in the case of repeated calculations.

It was found necessary to introduce artificial delays between the successive deliveries of the first letters to the NPs (see step 7 in phase 3), in order to limit the significant bus contention which would otherwise take place during step 3 of this phase. If the delays are removed, the demonstration package becomes less reliable [26–28].

C.2.5 Phase 5: Distributed Signal Processing

The distributed FFT or, more generally, the distributed operation at hand, is executed during this phase. Whereas the previous phases are largely independent of the particular application at hand, phase 5 is specifically tailored to the 1–D FFT. Even so, the general strategy and the critical pieces of software can easily be adapted to a number of other applications.

C.2.5.1 TFM

In this phase the role of the TFM is essentially limited to the monitoring of the distributed operations carried out in the nodes. The TFM collects task–completion messages through the mailboxes, and subsequently acknowledges them. In some instances, the message from a given NP is acknowledged immediately after arrival in the mailbox and subsequent detection by the TFM. In this case, the NP does not wait for the acknowledge before continuing with its next task; it will not, however, put any new message in the mailbox as long as the previous message has not been acknowledged by the TFM. The completion messages falling in this category are identified on the time–line with a "NO SYNCHR" label, and help the TFM to keep track of the progress accomplished by each NP. Under normal circumstances, these messages are not really required for a flawless execution of the application program. They are, however, extremely useful for debugging purposes in a developmental phase, and could certainly be exploited for reconfiguration purpose in the presence of true internal faults.

The remaining messages are labelled with "SYNCHR" on the time–line and are highly critical because they prevent any slippage in epoch . To achieve this, the TFM waits until all corresponding completion messages have been received from all NPs before acknowledging any of them. On the other hand, each NP waits for an acknowledge before proceeding. As will be seen on the time–line, synchronization through messages takes place at the epoch boundaries, i.e., after the "internal" butterfly calculations, and after each of the "external" butterfly stages.

Besides these communication and synchronization tasks, the TFM also resets the LFL bit and issues a hardware trigger to the NPs at the appropriate times.

C.2.5.2 NPs

The NPs perform the following arithmetic and transfer operations.

1. Load, on a *logical* identity basis, the appropriate transfer tables from load–and–test–memory to private TFM memory.

2. Load, on a *logical* identity basis, the appropriate portions of data and reference results from load–and–test–memory to private TFM memory.[13]

3. Report to the TFM that the loading from load–and–test memory is completed, and then proceed to the next task without delay. As far as the TFM is concerned, these task completion messages constitute the first clue that the NPs have effectively received their first letter (in phase 3), thereby allowing the TFM to reset the LFL bit.[14]

4. Wait for reception of the hardware trigger sent by the TFM through a parallel port. Even though using and waiting for the trigger is not required, this constitutes a means of starting all NPs, as simultaneously as possible, in their actual distributed processing operations.

5. Compute all stages of internal butterflies, report to the TFM that the current epoch (#1) is completed, and wait for the step–locking acknowledge, which indicates that all NPs are ready for the next epoch.

---

[13] The reason for decoupling steps 1 and 2 will become clear later.

[14] This facilitates recovery in case of abnormal termination. All processors are held in Modified Versabug mode as a result of this precautionary measure.

6. Configure the network for the first series of interprocessor transfers, report to the TFM, and proceed to the next task without waiting.

7. Transfer intermediate results over the network, report to the TFM, and proceed to the next task without waiting.

8. Compute the first stage of external butterflies, report to the TFM that the current epoch (#2) is completed, and wait for the step–locking acknowledge indicating that all NPs are ready.

9. Configure the network for the second series of interprocessor transfers, report to the TFM, and proceed to the next task without waiting.

10. Transfer intermediate results over the network, report to the TFM, and proceed to the next task without waiting.

11. Compute the second stage of external butterflies, report to the TFM that the current epoch (#3) is completed, and wait for the step–locking acknowledge indicating that all NPs are ready.

12. Rearrange the results from pseudo–normal to normal order by using the network twice. The successive steps involved are very similar to the steps 9, 10, 6, and 7 taken in that order, but have been deleted from the time–line for conciseness.

13. Check the NP results against the corresponding portion of precomputed results (previously crossloaded from load–and–test memory), and/or transfer the local results to load–and–test memory, report to the TFM, and then proceed to the next task without delay.

As explained earlier, the user may want to demonstrate either the distributed calculation of a single FFT, or simulate the operation of a real system where successive frames of data are loaded at regular intervals and then transformed. In the first case, the next operations are those of phase 6 (see below). To emulate the second, one can continuously reload the same data set from the load–and–test memory and compute its FFT. This is easily achieved by branching back to step 2 in the current phase. Furthermore, if an externally–injected fault is detected anywhere in phase 5, the TFM reconfigures the system, and the distributed processing resumes with step 1.

C.2.6 Phase 6: Termination

Each NP reports to the TFM that it has reached the end of its distributed signal processing operations (either after a single or a larger number of FFTs), and it subsequently returns to the Modified Versabug mode. After collecting all related messages, the TFM also returns to that mode. At this point, the state of the system is, for all practical purposes, identical to that of phase 3, just after the download. The demonstration program can be re–run simply by executing the TFM program again.

Appendix D

TRANSFER TABLES

D.1 INTRODUCTION

There are a large number of ways to calculate a multi–processor FFT. Section 4.6 describes the approach that was chosen for the demonstration program, but it is not unique. To accommodate alternative choices early in the software development cycle, a general transfer strategy based upon look–up tables was selected. This approach would not be chosen for an operational system because it is rather slow and requires significant memory space for the tables. However, the look–up table approach is quite general and could handle the transfers of many distributed calculations, without any modification to the assembly language data transfer subroutines.[15] Thus it is a reasonable choice for proof–of–concept experiments.

D.2 TERMINOLOGY

Figure 32–a indicates the data transfer that would occur for two processors executing an eight–point FFT. Unfolding the diagram yields Figure 32–b, or more generally, the "triplet" configuration of Figure 32–c. In writing programs for a distributed system, it is often convenient to imagine oneself located in some node, referred to as the *local processor,* and receiving data from the *source processor* while sending data to the *destination processor.* For all FFT transfers, the source and destination are always the same processor, but for other algorithms, this may not be the case.

---

[15] Strictly speaking, this is only true for situations not involving broadcast.

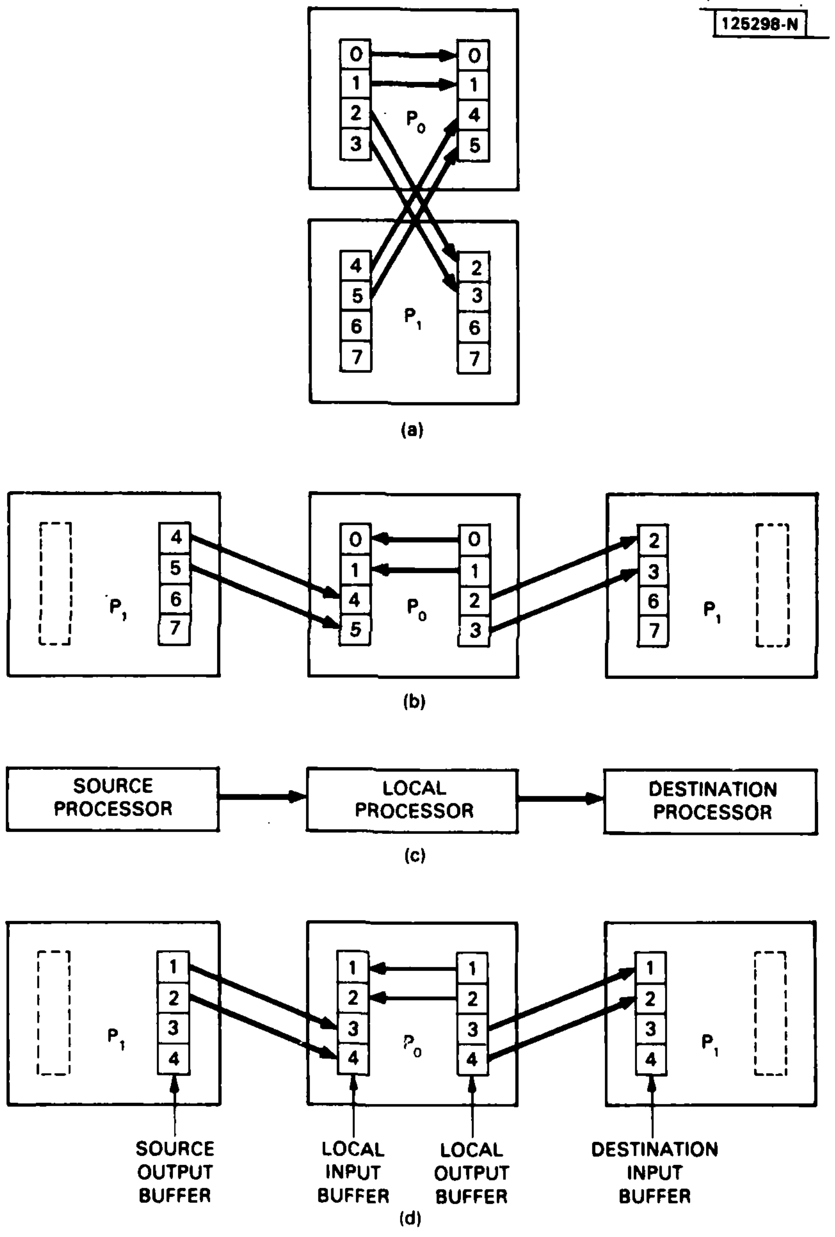


Fig. 32. Transfer Table Example

D.3 TRANSFER TABLE STRUCTURE

Before describing these tables, it is convenient to redraw Figure 32–b as shown *in Figure 32–d, where the single–processor memory* locations (which are irrelevant to the transfer themselves) have been replaced by indices identifying locations in the input and output buffers of interest in the source, local, and destination processors.

The transfer table for a given processor and a given transfer is shown in Table 9. The parameters SRC and DST identify the source and destination processors. The array SM(i) (Source Map) indicates where the ith element of the source output buffer goes in the local input buffer (a zero indicates that the related element does not leave the source processor). Similar interpretations can be given for LM(i) (Local Map) and DM(i) (Destination Map). To illustrate, the table corresponding to the transfer shown in Figure 32–d is shown in Table 10.

TABLE 9

FORMAT OF TRANSFER TABLE

SRC,DST	
SM(i)	
LM(i)	i=1,2,...N/P
DM(i)	

TABLE 10

TRANSFER TABLE EXAMPLE

```
   1, 1
3, 4, 0, 0
1, 2, 0, 0
0, 0, 1, 2.
```

This is all P_0 needs to handle the data transfer. Furthermore, with the exception of the SRC and DST parameters, the tables are identical for all transfers and all processors.

D.4 SOFTWARE

From Figure 32–d it is clear that the transfer operations at each node can be decomposed into three concurrent tasks: (1) the transfers from the source, (2) the internal rearrangements, and (3) the transfers to the destination. These tasks are handled by a subroutine and two interrupt handlers.

The subroutine initiates the first transfer to the destination and then proceeds to the internal rearrangements. Acknowledgements from the destination and requests from the source reach the local processor in the form of interrupts, which are processed by the first interrupt handler. Each byte transfer from the source or to the destination involves a full handshake via interrupts, as well as the use of a timer. If a new request or an acknowledge is not received within a prescribed period of time, additional interrupts are used to communicate the timeout information to the local processor, where they are processed by the second interrupt handler. Therefore, in the midst of a transfer, four different devices may, at any time, interrupt the internal rearrangement, i.e., the input and output ports, and two of the on–board timers.[16] This happens simultaneously in the four nodes, so that a total of eight ports and eight timers are likely to interrupt various parts of the distributed processor at any time.

Since each byte transfer is accompanied by a full, timed handshake, a timer must be started and stopped four times and four interrupts must be processed for each complex word. In addition, after completion of all transfers and rearrangements, a final transfer of the full input buffer into the output buffer is needed prior to the external butterfly calculations and the next transfer. This is also carried out by the above mentioned subroutine, which waits for the end of the interrupt–driven transfers before doing this final internal move.

---

[16] Detailed information on the use of the parallel ports and timers will be found in [29–32].

Appendix E

MAILBOXES

The distributed FFT calculation relies on the exchange of information between the TFM and each of the NPs, especially on the proper synchronization of all NPs by the TFM to prevent any slippage of epoch. This is achieved through letters deposited and picked-up by the various processors in a set of "mailboxes" located in load-and-test memory, where they can be accessed by all processors. Each NP (including the spare) possesses its own private mailbox (MBX), which consists of eight contiguous 16-bit words of memory.

Two options were available for the assignment of MBXs to the various processors. The MBXs could indeed be assigned either on the basis of physical identities (static assignment) or on the basis of logical identities (dynamic assignment). In the first case, a given set of 16 bytes of memory is allocated once and for all to a specific processor board. The drawback with this procedure is that the user and the TFM must continuously know the current relationship between logical and physical identities, in debugging mode and upon detection of an internal fault, respectively. In the second case, the same set of memory locations is assigned to the processor having a specific *logical* identity. This makes it easy for both the user and the TFM to interpret the MBX contents, but this requires special care at reconfiguration time to avoid conflicts when the MBX assignment changes.

The logical assignment has been adopted since it greatly simplifies the debugging and the error analysis by the TFM, because the MBX contents can then be interpreted independently of the current system configuration. With this choice the first MBX is always assigned to the processor with current logical address zero and the last one to the current spare (Figure 33).

The format of each MBX, and consequently of each letter, is shown in Figure 34. The single most important part of the MBX/letter is the first 16-bit word. The seven remaining words contain parameters which are used only in the presence of internal faults. These parameters have been found useful in the integration phase to reconstruct the series of events which caused the improper termination of a run. For the same reason, they would greatly help the TFM in its analysis of system status upon discovery of one of more internal faults.[17]

The most significant bit (MSB) of each MBX is a semaphore (S) which, in conjunction with the special purpose "Test-and-Set" (TAS) instruction of the MC68000, avoids any conflict or confusion in handling the MBX content. More specifically, one wishes to avoid the situation where two or more processors read, write, or test the same memory location, without one being able to tell with absolute certainty the final status of that location. Without the above mechanism (Semaphore plus TAS

---

[17] These parameters are not examined by the current version of the TFM program, which deals only with external faults.

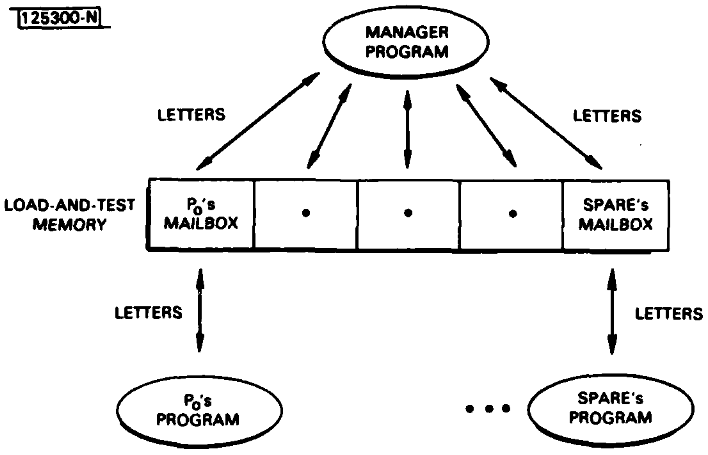


Fig. 33. Communication and Synchronization Mailboxes

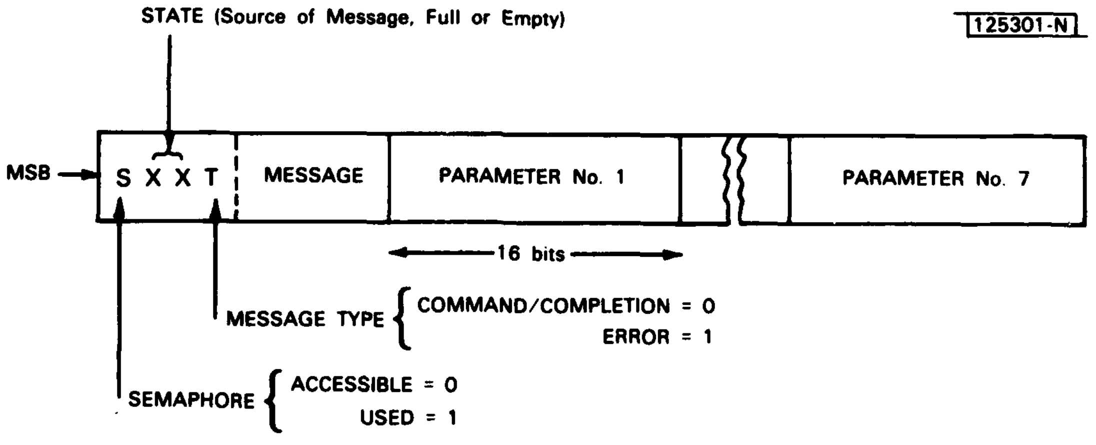


Fig. 34. Mailbox Format

instruction), confusion is bound to arise since in general the TFM and one or two NPs[18] compete for access to the same MBX, virtually at the same time.

To access a MBX, a program uses the "indivisible" instruction TAS which reads the first byte of the MBX, sets the MSB (S) and displays its previous state through the condition code. If S was previously set, the MBX is clearly being used by a competing

[18] This only happens during reconfiguration.

processor and should not be disturbed, otherwise it is available to the processor of interest, but not to the others. If one always uses the TAS instruction to read, write, or test the content of a MBX, no confusion will ever arise. However, the semaphore is not a hardware lock and a strict software discipline is required if one wishes to use it properly.

The next two bits of each MBX specify the state of the MBX (code XX). The first bit specifies the source of the current letter (0=TFM, 1=NP). If the second bit is equal to the first the MBX is "full", otherwise it is "empty". More specifically, the convention for the MBX code XX is as follows:

1. XX=00 indicates that the last letter was put in the MBX by the TFM and has yet to be read by the NP

2. XX=01 indicates that the last letter was put in the MBX by the TFM and has already been read by the NP

3. XX=11 indicates that the last letter was put in the MBX by the NP and has yet to be read by the TFM

4. XX=10 indicates that the last letter was put in the MBX by the NP and has already been read by the TFM

Therefore, after accessing a MBX through proper handling of the semaphore, a processor can determine the state of the MBX and act accordingly. For example, if a NP has previously reported that the network was successfully configured and finds a code XX=11 in its MBX while attempting to report the end of the corresponding transfer, it is clear that it must wait until the TFM has read the previous message.

While the next bit (T) in each MBX distinguishes the error messages from the others (0=command/completion message, 1=error message), the remaining 12 bits are used to identify a specific message among these two classes. A message with T=0 and a given 12-bit number may be either a command or a completion message. These are distinguished through the first bit of the code XX. Indeed, command messages are always issued by the TFM (XX=0?), whereas completion messages are always issued by the NPs (XX=1?).

The demonstration package comprises 4 command messages, 7 completion messages, and 17 error messages. The first letter sent by the TFM to the NPs is the only command message mentioned on the multi-processor timeline of Appendix C, since the others show up at reconfiguration time only. The completion messages, however, all appear at various points on the timeline. The error messages have, without exception, appeared at one time or another in the MBXs during development and have greatly eased the debugging. Similarly, they should be fully exploited to reconfigure the system upon discovery of some internal fault(s).[19] For a detailed list of all the messages, including their parameters, the reader is referred to the detailed flowcharts given in [25].

---

[19] This is not done in the current version of the demonstration package which only deals with externally-injected faults.

Appendix F

RECONFIGURATION

Upon detection of a manually–injected fault, a NP immediately reports the fact to the TFM by depositing an error message in its MBX. At that time, the TFM is probably busy acknowledging messages announcing the completion of a given task, such as the data transfers or the external butterfly calculations. Furthermore, the error message(s) is discovered only as the TFM looks at the MBX of the faulty processor(s) in its attempt to find and acknowledge the current task–completion message. As a result, if the error message is sent before the faulty NP has deposited the completion message currently expected by the TFM, the error message will rapidly be discovered by the TFM. Otherwise, it will not be discovered until the next round of acknowledgments.

Furthermore, some task completion messages cause the processors to be synchronized while others do not. The forced synchronization case is the simplest, since none of the NPs are allowed to proceed further until they have all reported. If one or more processors report an error, the situation is frozen and thus well defined .

If the message does not force synchronization, some of the NPs may already be engaged in subsequent operations when an error is discovered by the TFM. In this case, the situation is more complicated, and true internal faults are even likely to develop for timeout reasons if the subsequent operations involve data transfers among processors. These problems could be avoided by requiring synchronization each time a message is sent to the TFM, but this would probably degrade the overall performance, since the processors would be forced to wait when there would otherwise be no need to do so.

Note that the precise behavior of the TFM upon almost–simultaneous injection of two or more faults depends upon the exact timing of the injections. The TFM may (a) either quit immediately, or (b) reconfigure upon seeing a first fault, and then quit after detecting a second fault.

Thus, after finishing a complete series of acknowledges and discovering one or more error messages, the TFM analyzes the situation based solely on the MBX contents. In the meantime, the "faulty" processor(s) has transferred control to a reconfiguration subroutine where it await instructions from the TFM, while the "healthy" processors are waiting for an acknowledge which will never arrive as a result of the fault(s) just discovered by the TFM.

The error analysis by the TFM is straightforward in the case of externally–injected faults. This analysis is the only part of the reconfiguration procedure which would need to be refined and expanded to deal with all the errors which could be reported by each node in the case of an actual failure. Currently, the TFM does not attempt to reconfigure the system if more than one (external) fault is discovered. If only one fault has been reported, the TFM interrogates the spare through a command message deposited in its MBX to find out if it is usable, i.e., if its fault–injection thumbwheel is still in its original position.[20] If the current spare is found "healthy", the TFM then completes the system reconfiguration. Otherwise, in the case of multiple faults, it commands all NPs to quit via a MBX message and to return to Modified Versabug mode.

In the case of a single externally–injected fault, the processor which is found "faulty" becomes the new spare.[21] To perform the reconfiguration, the TFM determines the physical identity (ID) of the faulty processor which, so far, is known only through its logical ID as a result of the fact that MBXs are assigned on the basis of logical IDs. Since the only piece of information kept in memory by the TFM is the physical ID of the current spare, the TFM establishes the current relationships between logical and physical IDs and then, based on the logical ID of the faulty processor, determines its physical ID. This is also the physical ID of the new spare which is used later to set the sparing switches. The mapping between future logical and physical addresses is performed next. Finally, the mapping between current and future logical addresses is created.

Figure 35 illustrates the case where the current spare is the processor with physical ID "1" and a fault is reported by the processor with logical ID "2". The figure shows (a) the current logical–to–physical ID mapping, (b) the future logical–to–physical ID mapping, and finally (c) the current logical–to–future logical ID mapping.

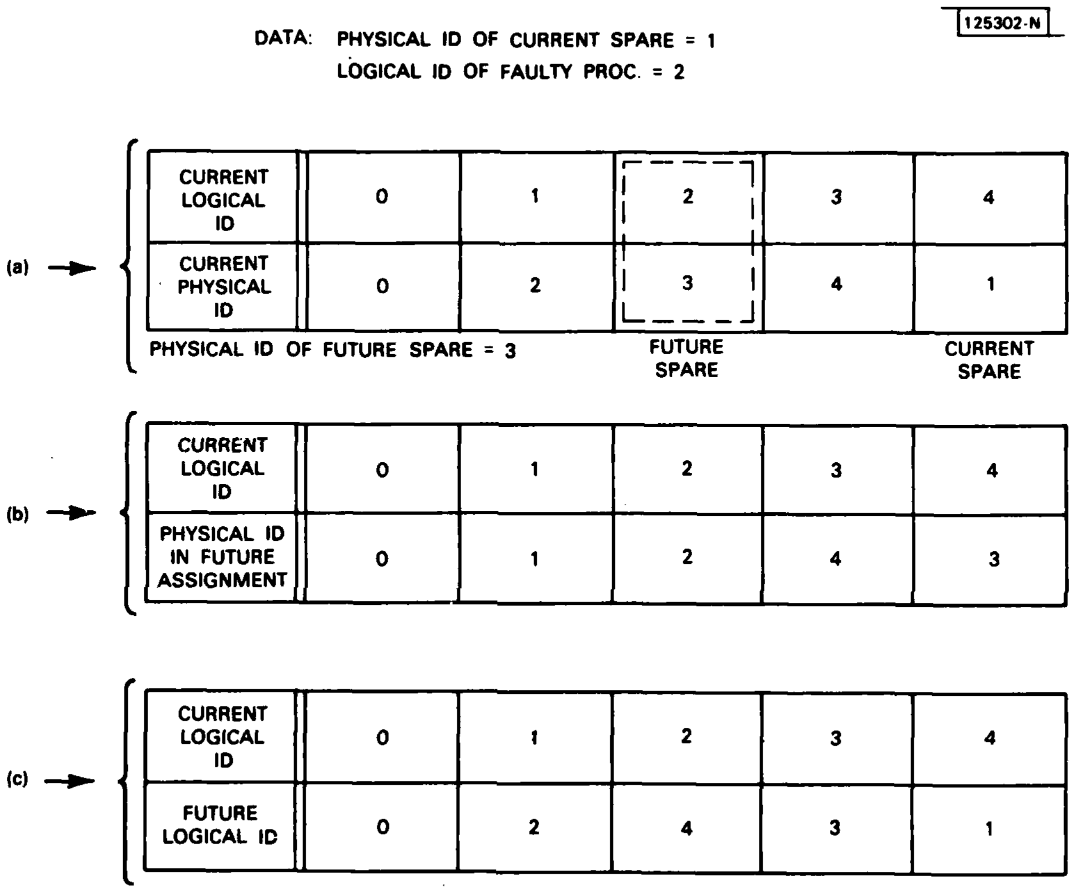
After these preliminary calculations, the TFM proceeds with the actual reconfiguration. The value of the future spare is used to set the sparing switches, while the assignment–transition vector is used to deposit in each MBX a message containing the new logical ID. After sending these reconfiguration commands to all processors (including the "old" and "new" spares), the TFM returns to its task of monitoring the DISP activities. In the meantime, each NP releases it currents MBX, waits for its new MBX to become available, and then recomputes a few pointers, reloads the appropriate transfer tables and finally resumes its endless calculation of FFTs .

In conclusion, the reconfiguration procedure is probably the most delicate part of the software because of the large variety of unpredictable situations that may occur.

---

[20] This is necessary to cover the situation where the current spare is the result of a previous reconfiguration, and also the unlikely case where a fault would have been inadvertently injected in the spare!

[21] This would not necessarily be the case in an actual system failure.



125302-N

DATA: PHYSICAL ID OF CURRENT SPARE = 1
LOGICAL ID OF FAULTY PROC. = 2

(a)

CURRENT LOGICAL ID	0	1	2	3	4
CURRENT PHYSICAL ID	0	2	3	4	1
			FUTURE SPARE		CURRENT SPARE

PHYSICAL ID OF FUTURE SPARE = 3

(b)

CURRENT LOGICAL ID	0	1	2	3	4
PHYSICAL ID IN FUTURE ASSIGNMENT	0	1	2	4	3

(c)

CURRENT LOGICAL ID	0	1	2	3	4
FUTURE LOGICAL ID	0	2	4	3	1

Fig. 35. Reconfiguration Example

Appendix G

UTILITY SOFTWARE

G.1 MODIFICATIONS TO MOTOROLA VERSABUG

Each Versamodule in the test–bed was equipped with a Versabug Debugger firmware module described in Section 4.8. Versabug runs in a tight command input loop awaiting user commands. For the test–bed, this loop was modified to examine the upper byte of parallel port 2 to determine whether the demonstration package is being exercised. With a positive indication, the modified Versabug takes extraordinary actions depending on its own identity (timing and fault manager or nodal processor, including spare) in accordance with the demonstration program.

Because Versabug is in firmware (eight 2K x 8 EPROMS occupying 16 Kbytes of address space (F00000 to F03FFF)), modifications to the program required new EPROM generation. The Versabug firmware is configured as 4 pairs of "odd–even" byte EPROMs. Thus, to patch in a jump to unused Versabug address space required a change to two EPROMs (an odd byte–even byte pair).

The Versabug modification was developed and assembled with the cross assembler. The resulting object code module was then run through a format conversion program described below, whose outputs were an "odd–byte" file and an "even–byte" file. These files were used to generate new EPROMS containing the extended Versabug code. This same technique was used modify the Versabug "upload" command to make it compatible with the existing 470 protocol.

G.2 ADJUNCT PROGRAMS

A wide variety of programming was carried out in support of the three major system programs; the TFM, the NP, and the multi–processor FFT subroutine. Some of these programs were written in MC68000 assembly language. Others were constructed using Fortran on the 470 and still others used the 470 CMS EXEC facilities.

One subset of these programs is assimilated directly into the demonstration package. Another subset is invoked by the demonstration program System as it operates. A final subset contains in–house utility programs operating outside of the main demonstration package.

G.2.1 Directly Assimilated Code

The Versabug Monitor/Debugger firmware is not designed to provide subroutines or program sequences for user programs operating on the VM01A. Nonetheless, certain useful functions can be invoked by object programs by using absolute addresses as entry points.

G.2.1.1 Display Messages on Terminal

This very useful facility is available to object programs by setting up an address register as a pointer to the message to be displayed and executing JSR (jump to subroutine) instructions directly into Versabug code.

G.2.1.2 Translucent Mode – Send Messages to Host

The "TM" (transparent mode) command in Versabug allows the user to communicate with the host machine. This is accomplished by Versabug acting as an intermediary between the two serial ports. Information coming from port 2 (470) is sent to port 1 (terminal) and vice versa.

By excising and modifying portions of Versabug, it is now possible for the object program to usurp the terminal intermediary role from Versabug and to conduct a dialogue with the host at the object code level in the VM01A. This mode is known as the "translucent mode". This feature adds tremendous flexibility to the test-bed since programs can now command the 470 to do virtually anything that a human operator can do at a terminal.

G.2.1.3 Upload Data to Host – Initiated at Object Code Level

One of the important functions of the test-bed is to display plots of the input data and processed data. As mentioned earlier, a modification to the standard Versabug has been made to permit the user at the terminal to upload data blocks to the host. A second subroutine extends this capability to the object program level.

G.2.1.4 Coefficient Table

The FFT program in the test-bed requires a table of coefficients in order to do the FFT calculations. The program requires the first-quadrant values of the cosine function based on a 256-point transform. The coefficient table is generated by a 470 Fortran program, and the outpu' is inserted into the MPFFT program using the CMS editor.

G.2.1.5 Transfer Tables

A technique somewhat similar to the "coefficient table" procedure was used to produce the "transfer tables". Recall from Appendix D that the transfer tables are used to control data transfers for the multi-processor FFT demonstration program. However they would not be used in an operational system because of their slow speed. The tables are generated on the 470 and stored in account DSP2 under the names:

DKTBL8A	DKTBL8B	(8-point FFT)
DKTBL16A	DKTBL16B	(16- " ")
DKTBL32A	DKTBL32B	(32- " ")
DKTBL64A	DKTBL64B	(64- " ")
DKTBL12A	DKTBL12B	(128- " ")
DKTBL25A	DKTBL25B	(256- " ")

The files whose last letter is A provide only the transfer tables necessary to complete the distributed FFT calculation. Hence, the data is left in pseudo-normal order. These "A" files are generated as follows.

1. Create the appropriate input data in the file named FILE FT04F001. (Consult the FORTRAN program DKMAPA to determine the necessary one line of data).

2. Run the FORTRAN program DKMAPA which produces the output file named FILE FT08F001.

3. Invoke the executive program DKMKFORM which modifies FILE FT08F001 to make it have the correct format for the desired M68K source file.

4. Rename FILE FT08F001 according to the convention described above.

5. Assemble the renamed source file to produce the object file which is used in the final load module.

The files whose last letter is B are a superset of the "A" files in that besides the information contained in the "A" files, the "B" files provide the extra transfer tables to rearrange the FFT results in normal order. These files are formed by following the same sequence of steps given above except that the FORTRAN program DKMAPB is used in step 2.

It should be noted that the two extra transfer stages included in DKMAPB will only rearrange into normal order the data from a four-processor FFT configuration. If more than four processors are used DKMAPB must be modified to include the appropriate additional transfer stages.

G.2.2 Invoked Code

The programs in this category do not execute in the VM01A. As the demonstration program is operating in the test–bed, the invoked–code programs are operating in the 470 host. This parallel form of system operation is made possible by the "translucent mode" described above.

At the appropriate places in the processing, the TFM sends a command to the host invoking the operation of a particular host program. The terminal output of the host program is then displayed on the operator's console with the demonstration package, in the translucent mode, acting as the serial port's intermediary.

The commands to the host consist of the invocation of one of two EXEC procedures. The two EXEC procedures are called by the TFM with arguments in the command line to specify a particular course of action for the EXEC procedure. The two EXECs, resident in the operator's 470 account, are called "PLOT EXEC" and "DRAW EXEC". (The PLOT EXEC is *not* the host system's same–named executive.)

G.2.2.1 PLOT EXEC

This EXEC is called in two places. First, it is called for plotting the input data set on the terminal, and second, for plotting the output (transformed) data set on the terminal. The distinction between which plot routine (input or output) is wanted by the system is made by an argument in the command line sent to the 470. The command sent is either 'PLOT PLTI <CR> <LF>' for input data plotting or 'PLOT PLTO <CR> <LF>' for output data plotting. PLTI and PLTO are absolute core–image files (modules) generated from two Fortran programs PLOTI and PLOTO written for each function (input or output). Having invoked the particular plot program, it remains for the TFM to call the "upload" subroutine to provide the data to the plot routine and thus produce the plot on the terminal.

G.2.2.2 DRAW EXEC

In addition to plotting the input and transformed data on the terminal, the demonstration program also shows a block diagram of the network configuration at various times (see Figures 22–24). The block diagrams are created by executing the appropriate subset of a set of fifteen programs on the host machine. These programs are written using the language of the host's BLOCK DIAGRAM PROGRAM. These programs are named according to the following convention

DKFxy

where

x = one of the integers 0, 1, 2, 3, 4

y = one of the letters A, B, C

The integer specified for x indicates which processor is shown as the spare by the block diagram. The letters A, B, and C correspond to

A : Configuration for internal butterflies (Epoch 1)

B : Configuration for external butterfly stage 1 (Epoch 2)

C : Configuration for external butterfly stage 2 (Epoch 3)

Note that the ordering of statements, while immaterial for hardcopy output, does make a difference when viewing the output of the program on a Tektronix terminal.

The way in which the block diagram facility is invoked parallels the "PLOT" technique described above. In this case the TFM sends a command line to the 470 of the form 'DRAW DKFxy <CR> <LF>' where the argument DKFxy is one of the fifteen files just mentioned. The DRAW EXEC then calls a 470 EXEC 'BLOCK', passing to it the argument just received from the demonstration program.

G.2.3 Utility Software

The day-to-day business of system development and enhancement was made somewhat easier by four utility programs described below.

G.2.3.1 JDED EXEC

The # (pound sign) character in the 470 environment is defaulted to mean 'line-end' or 'logical carriage return'. The MC68000 cross macro assembler uses the # to mean the immediate mode of addressing. The JDED EXEC changes the logical carriage return character to the % character in the 470, so the user is free to use # according to the assembler interpretation of that character. Additionally, JDED exec sets tab stops and then invokes the CMS editor with a filetype of M68K and a filename given to it as its argument. When editing is completed the # is restored to its usual 470 meaning.

G.2.3.2 JDMKSYS EXEC

When the user downloads the SYSTEM OBJECT file from the 470 into the TFM private RAM and into the load-and-test memory, five distinct pieces of code are loaded. The manager (TFM) program (JVMAN), the nodal processor program (JVNP), and the FFT program (DKFFT), go into the manager's private RAM while the data transfer tables and modified Versabug's RAM storage go into the load-and-test memory. Wherever a change is made to any of the five constituent programs, a new SYSTEM OBJECT file must be generated for downloading.

This chore is performed simply by invoking the exec file JMDKSYS which begins by asking the user for his choice of data transfer tables and then proceeds (with the help of the CMS editor) to create a new SYSTEM OBJECT file using the latest version of each of the constituent elements.

G.2.3.3 SYSCLR M68K

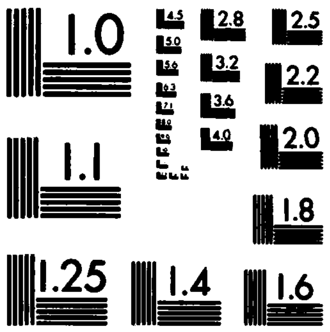
When power is first applied to the test–bed, the parity bit in every memory location may or may not cause a parity error during a RAM read. For this reason the SYSCLR program is executed before the first download after a power–on. SYSCLR is loaded into low RAM of the TFM where it proceeds to clear all of load–and–test memory as well as the TFM RAM from location 1000 (hexadecimal) to the top of memory. At this point system operation can safely proceed.

G.2.3.4 JDSTOPR FORTRAN

JDSTOPR is a Fortran utility program used to implement changes in the Versabug firmware (EPROMs).

AD-A130 227 A DISTRIBUTED SIGNAL PROCESSING ARCHITECTURE(U) 2/2
MASSACHUSETTS INST OF TECH LEXINGTON LINCOLN LAB
A E FILIP ET AL. 12 MAY 83 TR-637 ESD-TR-82-178
UNCLASSIFIED F19628-80-C-0002 F/G 9/2 NL

END
FILMED


1.0
4.5
5.0
5.6
6.3
7.1
8.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

REFERENCES

1. C. Clos, "A Study of Non–Blocking Switching Networks" *Bell System Technical Journal,* March, 1953, pp. 406–424

2. A. V. Oppenheim, R. W. Schafer, *Digital Signal Processing,* Prentice–Hall, 1975, pp. 307–314.

3. V. E. Benes, "Optimal Rearrangable Multi–stage Connecting Networks," *Bell System Techinal Journal,* July, 1964, pp. 1641–1656.

4. D. C. Opferman, N. T. Tsao–Wu, "On a Class of Rearrangable Switching Networks," *Bell System Technical Journal,* June, 1971, pp. 1579–1600.

5. D.H. Lawrie, "Access and Alignment of Data in Array Processor," *IEEE Trans. on Computers,* C–24, No. 12, December 1975, pp. 1145–1154.

6. K.E. Batcher, "The Flip Network in STARAN," *1976 Int. Conf. on Parallel Processing,* August 1976, pp. 65–71.

7. M.C. Pease, "The Indirect Binary *n*–Cube Microprocessor Array," *IEEE Trans. on Computers,* C–26, No. 5, May 1977, pp. 458–473.

8. T. Feng, "Data Manipulator Functions in Parallel Processors and their Implementations," *IEEE Trans. on Computers,* C–23, No. 3, March 1974, pp. 309–318.

9. T. Lang and H.S. Stone, "A Shuffle–Exchange Network with Simplified Control," *IEEE Trans. on Computers,* C–25, No. 1, January 1976, pp. 55–64.

10. L.R. Goke and G.J. Lipovski, "Banyan Networks for Partitioning Multiprocessor Systems," *Proc. First Annual Symposium on Computer Architecture,* December 1973, pp. 21–28.

11. C. Wu and T. Feng, "The Reverse–Exchange Interconnection Network," *1979 Conference on Parallel Processing,* pp. 160–175.

12. P.G. Jansen and J.W. Kessels, "The DIMOND: A Component for the Modular Construction of Switching Networks," *IEEE Trans. on Computers,* C–29, No. 10, October 1980, pp. 884–889.

13. R. Rettberg, et al, "Development of a Voice Funnel System: Design Report," Report No. 4098, Bolt, Beranek and Newman, Inc., August 1979.

14. D. Pradhan and K. Kodandapani, "A Uniform Representation of Single and Multistage Interconnection Networks Used in SIMD Machines," *IEEE Trans. on Computers,* C–29, No. 9, September 1980, pp. 777–791.

15. D. S. Parker, "Notes on Shuffle/Exchange-Type Switching Network," *IEEE Trans. on Computers,* C-29, No. 3, March 1980, pp. 213-222.

16. C. Wu and T. Feng, "On a Class of Multistage Interconnection Netorks," *IEEE Trans. on Computers,* C-29, No. 8, August 1980, pp. 694-702.

17. H.J. Siegel and S.D. Smith, "Study of Multistage Interconnection Networks," *1978 Symposium on Computer Architecture,* April 1978, pp. 223-229.

18. C. Wu and T. Feng, "Routing Techniques for a Class of Multistage Interconnection Networks," *1978 Conference on Parallel Processing,* pp. 197-206.

19. S. Thanawastien and V.P. Nelson, "Interference Analysis of Shuffle/Exchange Networks," *IEEE Trans. on Computers,* C-30, No. 8, August 1981, pp. 545-556.

20. J.M. Frankovich, "Bandwidth Analysis of Butterfly Networks," *Third International Symposium on Distributed Computing Systems,* 1982.

21. H.J. Siegel, "Analysis Techniques for SIMD Machine Interconnection Networks and the Effects of Processor Address Mask," *IEEE Trans. on Computers,* C-26, No. 2, February, 1977, pp. 153-161.

22. A.E. Filip, G.L. Kelly, and D.E. Kirk, "Distributed FFT Algorithms with Data Transfers Overlapping Computation," *16th Asilomar Conference on Circuits, Systems, and Computers,* November, 1982.

23. T. Bially, Private Communication.

24. D.B. Harris and J.H. McClellan, "Vector-Radix Fast Fourier Transform," *IEEE Int. Conf. on Acoustics, Speech and Signal Processing,* 1977, pp. 548-551.

25. J.G. Verly, "Demo-81 Flowcharts," Private Communication, 25 May 1982.

26. A.M. Pellegrini, "DISP Bus Contention Problem," Private Communication, 21 April 1982.

27. A.M. Pellegrini, "Results of Discussion with Motorola Regarding DISP Bus Contention," Private Communication, 27 April 1982.

28. A.M. Pellegrini, VERSAbus Arbitration Timing Analysis," Private Communication, 29 April 1982.

29. J.G. Verly, "On the Use of the Versamodule Parallel Ports," Private Communication, 16 June 1981.

30. J.G. Verly, "Experimenting with the Versamodule Parallel Ports," Private Communication, 16 June 1981.

31. J.G. Verly, "On the Use of the Versamodule Programmable Timer Module," Private Communication, 13 July 1981.

32. J.G. Verly, "Experimenting with the Versamodule Programmable Timer Module," Private Communication, 31 July 1981.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ESD-TR-82-178	2. GOVT ACCESSION NO. AD-A	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A Distributed Signal Processing Architecture		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER Technical Report 637
7. AUTHOR(s) Anthony E. Filip, James S. Arthur, John D. Drinan, Albert H. Huntoon, Donald E. Kirk, Jacques G. Verly		8. CONTRACT OR GRANT NUMBER(s) F19628-80-C-0002
9. PERFORMING ORGANIZATION NAME AND ADDRESS Lincoln Laboratory, M.I.T. P.O. Box 73 Lexington, MA 02173-0073		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Program Element No. 63428F Project No. 2698
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Systems Command, USAF Andrews AFB Washington, DC 20331		12. REPORT DATE 12 May 1983
		13. NUMBER OF PAGES 102
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Electronic Systems Division Hanscom AFB, MA 01731		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

signal processing
distributed processing
multi-processor
computer architecture
computer networks

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

An architecture is described for a multi-processor implementation of real-time signal processing algorithms. A "butterfly" network is used to provide simultaneous, conflict-free interprocessor communication for multi-dimensional convolution and Fourier transformation. A hardware demonstration test-bed using four active processors was used to validate the concepts of (1) shared algorithm execution, (2) conflict-free data transfers, (3) distributed network control, (4) dynamic fault tolerance, and (5) identical software in all processors.

DD FORM 1473 1 Jan 73 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)